AFRL-VA-WP-TM-2005-3080

COMPOSITIONAL ABSTRACTION AND REFINEMENT FOR ASPECTS (CARA)

Dr. John Rushby

SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025-3493

MARCH 2004

Final Report for 01 July 2000 – 01 March 2004

Approved for public release; distribution is unlimited.

STINFO FINAL REPORT

AIR VEHICLES DIRECTORATE
AIR FORCE MATERIEL COMMAND
AIR FORCE RESEARCH LABORATORY
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Wright Site Public Affairs Office (AFRL/WS) and is releasable to the National Technical Information Service (NTIS). It will be available to the general public, including foreign nationals.

PAO Case Number: AFRL/WS 05-1124, 09 May 2005.

THIS TECHNICAL REPORT IS APPROVED FOR PUBLICATION.

/s/

Raymond A. Bortner
Senior Electronic Engineer

/s/

Michael P. Camden, Chief
Control Systems Development and
Applications Branch
Control Sciences Division

/s/

Brian W. Van Vliet
Chief, Control Sciences Division
Air Vehicles Directorate

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
March 2005	Final	07/01/2000– 03/01/2004

4. TITLE AND SUBTITLE
COMPOSITIONAL ABSTRACTION AND REFINEMENT FOR ASPECTS (CARA)

5a. CONTRACT NUMBER
F33615-00-C-3043

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
0602301

6. AUTHOR(S)
Dr. John Rushby

5d. PROJECT NUMBER
A04Z

5e. TASK NUMBER

5f. WORK UNIT NUMBER
AL

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025-3493

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Vehicles Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson Air Force Base, OH 45433-7542

Defense Advanced Research Projects Agency (DARPA/IXO)
3701 N. Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/VACC

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-VA-WP-TM-2005-3080

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Report contains color.

14. ABSTRACT

The project originally focused on compositional formal methods for aspect-oriented programs and was located in the PCES program. Soon after its inception, however, the project was moved to the SEC (Software Enabled Control) program where its focus shifted to formal analysis of mixed discrete/continuous (i.e., hybrid) systems. We developed a two-step approach to analysis of hybrid systems: compute a property-preserving discrete approximation to the original hybrid system, and then analyze the discrete approximation. The approximation method is called *Hybrid Abstraction* and was developed by us in the DARPA MoBIES program. In the present project, we developed the theorem-proving technology that enables automated calculation of the approximation, and we built the SAL (Symbolic Analysis Laboratory) system for specification and analysis of discrete systems.

15. SUBJECT TERMS
Formal methods, aspect-oriented design, Hybrid systems, program composition

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Raymond A. Bortner
Unclassified	Unclassified	Unclassified	SAR	148	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-8292

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Part I

Introduction

1

2

This report covers the period July 1, 2000 through March 1, 2004 and documents work performed by SRI International for the DARPA PCES and SEC programs through AFRL-WPAFB Contract F33615-00-C-3043.

The project originally focused on compositional formal methods for aspect-oriented programs and was located in the PCES program. Soon after its inception, however, the project was moved to the SEC (Software Enabled Control) program where its focus shifted to formal analysis of mixed discrete/continuous (i.e., hybrid) systems. We developed a two-step approach to analysis of hybrid systems: compute a property-preserving discrete approximation to the original hybrid system, and then analyze the discrete approximation. The approximation method is called *Hybrid Abstraction* and was developed by us in the DARPA MoBIES program. In the present project, we developed the theorem-proving technology that enables automated calculation of the approximation, and we built the SAL (Symbolic Analysis Laboratory) system for specification and analysis of discrete systems.

Research Products

The outputs of this research are documented in a series of technical reports and papers that are collected in Part II of this report. Below, we provide an index and abstracts for these papers. All the papers were selected for presentation at major scientific conferences, and we also provide citations for these publications.

In addition, the methods developed in this project were implemented in prototype tools. DARPA reduced the funding and scope of this project during its execution, and it was terminated early. Nonetheless, we were able to produce the first prototype of the SAL system (SAL 1.0) and to make this available to the research community. With funding from other sources, we have been able to continue development of SAL and this is now a robust and capable system with many users. SAL is available for download from `http://sal.csl.sri.com`; a description of its current capabilities is provided in [1] and its successful application to a large problem is described in [2].

Little Engines of Proof by N. Shankar. Published as [3].

> The key to practical computation of the approximations used in Hybrid Abstraction is efficient theorem proving over a combination of arithmetic theories. The approach used in this project is based on constructing decision procedures for individual theories, and then combining them to yield a decision procedure for the combined theory. This approach, now widely adopted, is advocated in this influential paper under the name "little engines of proof."

Abstract The automated construction of mathematical proof is a basic activity in computing. Since the dawn of the field of automated reasoning, there have been two divergent schools of thought. One school, best represented by Alan Robinson's resolution method, is based on simple uniform proof search procedures guided by

heuristics. The other school, pioneered by Hao Wang, argues for problem-specific combinations of decision and semi-decision procedures. While the former school has been dominant in the past, the latter approach has greater promise. In recent years, several high-quality inference engines have been developed, including propositional satisfiability solvers, ground decision procedures for equality and arithmetic, quantifier elimination procedures for integers and reals, and abstraction methods for finitely approximating problems over infinite domains. We describe some of these "little engines of proof" and a few of the ways in which they can be combined. We focus in particular on combining different decision procedures for use in automated verification.

Deconstructing Shostak By Harald Rueß and N. Shankar. Published as [4].

An important technique for combining "little engines of proof" was originally developed at SRI in the 1970s by Robert Shostak. Although widely used, the foundations of this method have not been rigorously established and prior to this paper, all treatments and implementations were flawed.

Abstract Decision procedures for equality in a combination of theories are at the core of a number of verification systems. Shostak's decision procedure for equality in the combination of solvable and canonizable theories has been around for nearly two decades. Variations of this decision procedure have been implemented in a number of systems including STP, Ehdm, PVS, STeP, and SVC. The algorithm is quite subtle, and a correctness argument for it has remained elusive. Shostak's algorithm and all previously published variants of it yield incomplete decision procedures. We describe a variant of Shostak's algorithm along with proofs of termination, soundness, and completeness.

Verifying Shostak by Jonathan Ford and N, Shankar. Published as [5].

This paper confirms the correctness of the argument developed in the previous paper by formally verifying it using SRI's PVS system.

Abstract Decision procedures for combinations of theories are at the core of many modern theorem provers such as ACL2, Ehdm, PVS, SIMPLIFY, the Stanford Pascal Verifier, STeP, SVC, and Z/Eves. Shostak, in 1984, published a decision procedure for the combination of canonizable and solvable theories. Recently, Rueß and Shankar showed Shostak's method to be incomplete and nonterminating, and presented a correct version of Shostak's algorithm along with informal proofs of termination, soundness, and completeness. We describe a formalization and mechanical verification of these proofs using the PVS verification system. The formalization itself posed significant challenges and the verification revealed some gaps in the informal argument.

Combining Shostak Theories by N. Shankar and Harald Rueß. Published as [6]

Shostak's method works for theories that are canonizable and solvable. The combination of the canonizers yields a canonizer for the combination, but this is not the case for solvers. This paper presents a crucial extension to Shostak's method that resolves this difficulty.

Abstract Ground decision procedures for combinations of theories are used in many systems for automated deduction. There are two basic paradigms for combining decision procedures. The Nelson-Oppen method combines decision procedures for disjoint theories by exchanging equality information on the shared variables. In Shostak's method, the combination of the theory of pure equality with canonizable and solvable theories is decided through an extension of congruence closure that yields a canonizer for the combined theory. Shostak's original presentation, and others that followed it, contained serious errors that were corrected for the basic procedure by the present authors. Shostak also claimed that it was possible to combine canonizers and solvers for disjoint theories. This claim is easily verifiable for canonizers, but is unsubstantiated for the case of solvers. We show how our earlier procedure can be extended to combine multiple disjoint canonizable, solvable theories within the Shostak framework.

On the Confluence of Linear Shallow Term Rewrite Systems by Guillem Godoy, Ashish Tiwari, and Rakesh Verma. Available as [7].

The method for computing the approximation used in Hybrid Abstraction uses insights from the papers above and from this one. The culmination of all these techniques is the method used in HybridSAL, which was funded under the MoBIES program and is described in [8].

Abstract This paper shows that the confluence of shallow linear term rewrite systems is decidable. This class of rewrite systems properly includes ground rewrite systems and shallow, linear, and nonsharing rewrite systems for which confluence was shown to admit a polynomial time decision procedure previously. For example, the commutativity axiom falls under this class. The decision procedure presented in this paper is a nontrivial generalization of the polynomial time algorithms for deciding confluence of ground and restricted nonground term rewrite systems presented previously. This algorithm has a polynomial time complexity if the maximum arity of a function symbol in the signature is considered a constant. This paper also gives EXPTIME-hardness proofs for reachability and confluence of shallow term rewrite systems. This shows that the shallow linear assumptions made in this paper are fairly tight.

The SAL Language Manual by Leonardo de Moura, Sam Owre, and N. Shankar. Available as [9].

The heart of the SAL system is its language, also called SAL. The SAL language provides an attractive language for writing specifications, and it is also suitable as a target for translating specifications originally written in other notations.

Abstract SAL stands for Symbolic Analysis Laboratory. It is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. A key part of the SAL framework is a language for describing transition systems. This language serves as a specification language and as the target for translators that extract the transition system description for popular programming languages such as Esterel, Java, and Statecharts. The language also serves as a common source for driving different analysis tools through translators from the SAL language to the input format for the tools, and from the output of these tools back to the SAL language.

The SAL language was originally designed in collaboration with David Dill of Stanford University and Thomas Henzinger of the University of California at Berkeley. The version presented here is the one currently accepted by the tools developed at SRI.

A Technique for Invariant Generation by Ashish Tiwari, Harald Rueß, Hassen Saïdi, and N. Shankar. Published as [10].

Although the SAL tools currently available (see `http://sal.csl.sri.com`) are all model checkers, the larger plan includes construction of bridges to deductive methods such as PVS. An important technique in deductive verification is the method of inductive invariance, and a crucial element in the automation of this method is automated construction and strengthening of auxiliary invariants. This paper describes methods for accomplishing this task.

Abstract Most of the properties established during verification are either invariants or depend crucially on invariants. The effectiveness of automated formal verification is therefore sensitive to the ease with which invariants, even trivial ones, can be automatically deduced. While the strongest invariant can be defined as the least fixed point of the strongest post-condition of a transition system starting with the set of initial states, this symbolic computation rarely converges. We present a method for invariant generation and strengthening that relies on the simultaneous construction of least and greatest fixed points, restricted widening and narrowing, and quantifier elimination. The effectiveness of the method is demonstrated on a number of examples.

Bibliography

[1] Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. Submitted for publication, January 2004. Available at `http://www.csl.sri.com/~rushby/abstracts/sal-tool`.

[2] Wilfried Steiner, John Rushby, Maria Sorea, and Holger Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. Submitted for publication, December 2003. Available at `http://www.csl.sri.com/~rushby/abstracts/startup-verification`.

[3] Natarajan Shankar. Little engines of proof. In Lars-Henrik Eriksson and Peter Lindsay, editors, *Formal Methods Europe (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 1–20, Copenhagen, Denmark, July 2002. Springer-Verlag.

[4] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, Boston, MA, July 2001. IEEE Computer Society.

[5] Jonathan Ford and Natarajan Shankar. Verifying Shostak. In A. Voronkov, editor, *Automated Deduction—CADE-18, 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 347–362, Copenhagen, Denmark, July 2002. Springer-Verlag.

[6] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *International Conference on Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 1–18, Copenhagen, Denmark, July 2002. Springer-Verlag.

[7] Guillem Godoy, Ashish Tiwari, and Rakesh Verma. On the confluence of linear shallow term rewrite systems. In H. Alt and M. Habib, editors, *20th Intl. Symposium on Theoretical Aspects of Computer Science (STACS 2003)*, volume 2607 of *Lecture Notes in Computer Science*, pages 85–96. Springer-Verlag, February 2003.

[8] Ashish Tiwari. Abstraction based theorem proving: An example from the theory of Reals. In Silvio Ranise and Cesare Tinelli, editors, *Proceedings of the CADE-19*

Workshop on Pragmatics of Decision Procedures in Automated Deduction, PDPAR 2003, pages 40–52, Miami, FL, July 2003. The full proceedings are available at `http://www.cs.miami.edu/~geoff/CADE-19/W2.pdf`; this paper is also available at `http://www.csl.sri.com/users/tiwari/pdpar03.html`.

[9] Leonardo de Moura, Sam Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2001. Revised August 2003.

[10] Ashish Tiwari, Harald Rueß, Hassen Saïdi, and N. Shankar. A technique for invariant generation. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127, Genova, Italy, April 2001. Springer-Verlag.

Part II

Technical Papers

9

10

Invited paper for FLoC'02. Appears in the Proceedings of FME'02, LNCS.

Little Engines of Proof*

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (650) 859-5272 Fax: +1 (650) 859-2844

Abstract. The automated construction of mathematical proof is a basic activity in computing. Since the dawn of the field of automated reasoning, there have been two divergent schools of thought. One school, best represented by Alan Robinson's resolution method, is based on simple uniform proof search procedures guided by heuristics. The other school, pioneered by Hao Wang, argues for problem-specific combinations of decision and semi-decision procedures. While the former school has been dominant in the past, the latter approach has greater promise. In recent years, several high quality inference engines have been developed, including propositional satisfiability solvers, ground decision procedures for equality and arithmetic, quantifier elimination procedures for integers and reals, and abstraction methods for finitely approximating problems over infinite domains. We describe some of these "little engines of proof" and a few of the ways in which they can be combined. We focus in particular on combining different decision procedures for use in automated verification.

Its great triumph was to prove that the sum of two even numbers is even.

Martin Davis [Dav83] (on his Presburger arithmetic procedure)

The most interesting lesson from these results is perhaps that even in a fairly rich domain, the theorems actually proved are mostly ones which call on a very small portion of the available resources of the domain.

Hao Wang (quoted by Davis [Dav83])

* Funded by NSF Grants CCR-0082560 and CCR-9712383, DARPA/AFRL Contract F33615-00-C-3043, and NASA Contract NAS1-20334. John Rushby, Sam Owre, Ashish Tiwari, and Tomás Uribe commented on earlier drafts of this paper.

1 Introduction

At a very early point in its development, the field of automated reasoning took an arguably wrong turn. For nearly forty years now, the focus in automated reasoning research has been on *big iron*: general-purpose theorem provers based on uniform proof procedures augmented with heuristics. These efforts have not been entirely fruitless. As success stories, one might list an impressive assortment of open problems that have succumbed to semi-brute-force methods, and spin-off applications such as logic programming. However, there has been very little discernible progress on the problem of automated proof construction in any significant mathematical domain. Proofs in these domains tend to be delicate artifacts whose construction requires a collection of well-crafted instruments, little engines of proof, working in tandem. In other disciplines such as numerical analysis, computer algebra, and combinatorial algorithms, it is quite common to have libraries of useful routines. Such software libraries have not taken root in automated deduction because the scientific and engineering challenges involved are quite significant. We examine some of the successes in building and combining little deduction engines for building proofs and refutations (e.g., counterexamples), and survey some of the challenges that still lie ahead.

The tension between general-purpose proof search and special-purpose decision procedures has been with us from very early on. Automated reasoning had its beginnings in the pioneering Logic Theorist system of Newell, Shaw, and Simon [NSS57]. The theorems they proved were shown by Hao Wang [Wan60b] to fall within simply decidable fragments like propositional logic and the $\forall^*\exists^*$ Bernays-Schönfinkel fragment of first-order logic [BGG97]. Many technical ideas from the Logic Theorist such as subgoaling, substitution, replacement, and forward and backward chaining, have been central to automated reasoning, but the dogma that human-oriented heuristics are the key to effective theorem proving has not been vindicated. Hao Wang [Wan60a] proposed an entirely different approach that he called *inferential analysis* as a parallel to numerical analysis. Central to his approach was the use of domain-specific decision and semi-decision procedures, so that proofs could be constructed by means of reductions to some combination of problems that could each be easily solved. Due to the prevailing bias in artificial intelligence, Wang lost the debate at that point in time, but, as we argue here, his ideas still make plenty of sense. As remarked by Martin Davis [Dav83]:

> *The controversy referred to may be succinctly characterized as being between the two slogans: "Simulate people" and "Use mathematical logic". ... Thus as early as 1961 Minsky [Min63] remarked*
>
> > ... it seems clear that a program to solve real mathematical problems will have to combine the mathematical sophistication of Wang with the heuristic sophistication of Newell, Shaw, and Simon.

The debate between human-oriented and logic-oriented approaches is beside the point. The more significant debate in automated reasoning is between two approaches that in analogy with economics can be labelled as *macrological* and *micrological*. The macrological approach takes a language and logic such as first-order logic as given, and attempts to find a uniform (i.e., problem-independent) method for constructing proofs of conjectures stated in the logic. The micrological approach attacks a class of problems and attempts to find the most effective way of validating or refuting conjectures in this problem class. In his writings, Hao Wang was actually espousing a micrological viewpoint. He wrote [Wan60a]

> *In contrast with pure logic, the chief emphasis of inferential analysis is on the efficiency of algorithms, which is usually obtained by paying a great deal of attention to the detailed structure of problems and their solutions, to take advantage of possible systematic short cuts.*

Automated reasoning got off to a running start in the 1950s. Already in 1954, Davis [Dav57] had implemented a decision procedure for Presburger arithmetic [Pre29]. Davis and Putnam [DP60], during 1958–60, devised a decision procedure for CNF satisfiability (SAT) based on inference rules for propagation of unit clauses, ground resolution, deletion of clauses with pure literals, and splitting. The ground resolution rule turned out to be space-inefficient and was discarded in the work of Davis, Logemann, and Loveland [DLL62]. Variants of the latter procedure are still employed in modern SAT solvers. Gilmore [Gil60] and Prawitz [Pra60] examined techniques for first-order validity based on Herbrand's theorem. Many of the techniques from the 1950s still look positively modern.

Robinson's introduction [Rob65] of the resolution principle (during 1963–65) based on unification brought about a qualitative shift in automated theorem proving. From that point on, the field of automated reasoning never looked forward. Resolution provides a simple inference rule for refutational proofs for first-order statements in skolemized, prenex form. It spawned a multitude of strategies, heuristics, and extensions. Nearly forty years later, resolution [BG01] remains extremely popular as a general-purpose proof search method primarily because the basic method can be implemented and extended with surprising ease. Resolution-based methods have had some success in proving open problems in certain domains where general-purpose search can be productive. The impact of resolution on theorem proving in mathematically rich domains has not been all that encouraging.

The popularity of uniform proof methods like resolution stems from the simple dogma that since first-order logic is a generic language for expressing statements, generic first-order proof search methods must also be adequate for finding proofs. This central dogma seems absurd on the face of it. Stating a problem and solving it are two quite separate matters. But the appeal of the dogma is obvious. A simple, generic method for proving theorems basically hits the jackpot by fulfilling Leibniz's dream of a reasoning machine. A more sophisticated version

of the dogma is that a uniform proof method can serve as the basic structure for introducing domain-specific automation. There is little empirical evidence that even this dogma has any validity.

On the other hand, certain domain-specific automated theorem provers have been quite effective. The Boyer-Moore line of theorem provers [BM79,KMM00] has had significant success in the area of inductive proofs of recursively defined functions. Various geometry theorem provers [CG01] based on both algebraic and non-algebraic, machine-oriented and human-oriented methods, have been able to automatically prove theorems that would tax human ingenuity. Both of these classes of theorem provers owe their success to domain-specific automation rather than general-purpose theorem proving.

Main Thesis. Automated reasoning has for too long been identified with uniform proof search procedures in first-order logic. This approach shows very little promise. The basic seduction of uniform theorem proving techniques is that phenomenal gains could be achieved with very modest implementation effort. Hao Wang [Wan60b,Wan60a,Wan63] in his early papers on automated reasoning sketched the vision of a field of inferential analysis that would take a deeper look at the problem of automating mathematical reasoning while exploiting domain-specific decision procedures. He wrote [Wan63]

> *That proof procedures for elementary logic can be mechanized is familiar. In practice, however, were we slavishly to follow these procedures without further refinements, we should encounter a prohibitively expansive element. . . . In this way we are led to a closer study of reduction procedures and of decision procedures for special domains, as well as of proof procedures of more complex sorts.*

Woody Bledsoe [Ble77] made a similar point in arguing for semantic theorem proving techniques as opposed to resolution.

Decision procedures [Rab78], and more generally inference procedures, are crucial to the approach advocated here. Few problems are stated in a form that is readily decidable, but proof search strategies, heuristics, and human guidance can be used to decompose these problems into decidable subproblems. Thus, even though not many interesting problems are directly expressible in Presburger arithmetic, a great many of the naturally arising proof obligations and subproblems do fall into this decidable class.

Building a library of automated reasoning routines along the lines of numerical analysis and computer algebra, is not as easy as it looks. A theorem prover has a simple interface in that it is given a conjecture and it returns a proof or a disproof. The lower-level procedures often lack clear interface specifications of this sort. Even if they did, building a theorem prover out of modular components may not be as efficient as a more monolithic system. Boyer and Moore [BM86] indicate how even a simple decision procedure can have a complex interaction with the other components, so that it is not merely a black box that returns *proved* or

disproved. The construction of modular inference procedures is a challenging research issues in automated reasoning.

Work on little engines of proof has been gathering steam lately Many groups are actively engaged in the construction of little proof engines, while others are putting in place the train tracks on which these engines can run. PVS [ORSvH95] itself can be seen as an attempt to unify many different inference procedures: typechecking, ground decision procedures, simplification, rewriting, MONA [EKM98], model checking [CGP99], abstraction, and static analysis, within a single system with an expressive language for writing mathematics.

2 Propositional Logic

The very first significant metamathematical results were those on the soundness, completeness, and decidability of propositional logic [Pos21]. Since boolean logic has applications in digital circuit design, a lot of attention has been paid to the problem of propositional satisfiability. A propositional formula ϕ is built from propositional atoms p_i by means of negation $\neg\phi$, disjunction $\phi_1 \vee \phi_2$, and conjunction $\phi_1 \wedge \phi_2$. Further propositional connectives can be defined in terms of basic ones like $\neg$ and $\vee$. A propositional formula can be placed in *negation normal form*, where all the negations are applied only to propositional atoms. A literal l is an atom p or its negation $\neg p$. A clause C is a disjunction of literals. By labelling subformulas with atoms and using distributivity, any propositional formula can be efficiently transformed into one that is in conjunctive normal form (CNF) as a conjunction of clauses. A CNF formula can be viewed as a bag Γ of clauses. The Davis–Putnam method (DP) [DP60] consisted of the following rules:

1. Unit propagation: l, Γ is satisfiable if $\Gamma[l \mapsto \top, \neg l \mapsto \bot]$ is satisfiable.
2. Pure literal: Γ is satisfiable if $\Gamma - \Delta$ is satisfiable, for $\neg l \notin [\![\Gamma]\!]$, where $[\![\Gamma]\!]$ is the set of subformulas of Γ, and $l \in C$ for each $C \in \Delta$.
3. Splitting: Γ is satisfiable if either l, Γ or $\neg l, \Gamma$ is satisfiable.
4. Ground resolution: $l \vee C_1, \neg l \vee C_2, \Gamma$ is satisfiable if $C_1 \vee C_2, \Gamma$ is satisfiable.

The Davis–Logemann–Loveland (DLL) variant [DLL62] drops the ground resolution rule since it turned out to be space-inefficient. Several modern SAT solvers such as SATO [Zha97], GRASP [MSS99], and Chaff [MMZ+01], are based on the DLL method. They are capable of solving satisfiability problems with hundreds of thousands of propositional variables and clauses. With this kind of performance, many significant applications become feasible including invariant-checking for systems of bounded size, bounded model checking, i.e., the search for counterexamples of length k for a temporal property, and boolean equivalence checking where two circuits are checked to have the same input/output behavior.

Stålmarck's method [SS00] does not employ a CNF representation. Truth values are propagated from formulas to subformulas through a method known as saturation. There is a splitting rule similar to that of DP, but it can be applied to subformulas and not just propositions. The key component of Stålmarck's method is the dilemma rule which considers the intersection of the two subformula truth assignments derived from splitting. Further splitting is carried out with respect to this intersection.

Binary Decision Diagrams. Reduced Ordered Binary Decision Diagrams (ROBDDs) [Bry86] are a canonical representation for boolean functions, i.e., functions from $[B^n \rightarrow B]$. BDDs are binary branching directed acyclic graphs where the nodes are variables and the outgoing branches correspond to the assignment of $\top$ and $\bot$ to the variable. There is a total ordering of variables that is maintained along any path in the graph. The graph is kept in reduced form so that if there is a node such that both of its branches lead to the same subgraph, then the node is eliminated.

Standard operations like negation, conjunction, disjunction, composition, and boolean quantification, have efficient implementations using BDDs. The BDD data structure has primarily been used for boolean equivalence checking and symbolic model checking. The main advantage of BDDs over other representations is that checking equivalence is easy. Boolean quantification is also handled more readily using BDDs. BDDs can also be used for SAT solving since it is in fact a compact representation for all solutions of a boolean formula. But the strength of BDDs is in representing boolean functions of a low communication complexity, i.e., where it is possible to partition the variables so that there are few dependencies between variables across the partition. BDDs have been popular for symbolic model checking [CGP99] and boolean equivalence checking.

Quantified Boolean Formulas and Transition Systems. In a propositional logic formula, all variables are implicitly universally quantified. One obvious extension is the introduction of Boolean existential and universal quantification. The resulting fragment is called quantified boolean formulas (QBF). This kind of quantification can be expressed purely in propositional logic. For example, the formula $(\exists p : Q)$ is equivalent to $(Q[p \mapsto \top] \vee Q[p \mapsto \bot])$. The language of QBF is of course exponentially more succinct than propositional logic. The decision procedure for QBF validity is a PSPACE-complete problem. Many interesting problems that can be cast as interactive games can be mapped to QBF.

Finite-state transition systems can be defined in QBF. A finite state type consists of a finite number of distinct variables over types such as booleans, scalars, subranges, and finite arrays over a finite element type. A finite state type can be encoded in binary form. A transition system over a finite state type that is represented by n boolean variables then consists of an initialization predicate I that is an n-ary boolean function, and a transition relation N that is a $2n$-ary boolean function. The nondeterministic choice between two transition relations N_1 and N_2 is easily expressed as $N_1 \vee N_2$. Internal state can be hidden through

boolean quantification. The composition $(N_1; N_2)$ of two transition relations N_1 and N_2 can be captured as $\exists \overline{y} : N_1(\overline{x}, \overline{y}) \wedge N_2(\overline{y}, \overline{x}')$.

Fixpoints and Model Checking. QBF can be further extended through the addition of fixpoint operators that can capture the transitive closure of a transition relation. Given a transition relation N, the reflexive-transitive closure of N can be written as $\mu Q : \overline{x}' = \overline{x} \vee (\exists \overline{y} : N(\overline{x}, \overline{y}) \wedge Q(\overline{y}, \overline{x}'))$. Similarly, the set of states reachable from the initial set of state can be represented as $\mu Q : I(\overline{x}) \vee (\exists \overline{y} : Q(\overline{y}) \wedge N(\overline{y}, \overline{x}))$. The boolean function represented by a fixpoint formula can be computed by unwinding the fixpoint until convergence is reached. For this, the ROBDD representation of the boolean function is especially convenient since it makes it easy to detect convergence through an equivalence test, and to represent boolean quantification [BCM+92,McM93]. The boolean fixpoint calculus can easily represent the temporal operators of the branching-time temporal logic CTL where one can for example assert that a property always (or eventually) holds on all (or some) computation paths leading out of a state. The boolean fixpoint calculus can also represent different fairness constraints on paths. The emptiness problem for Büchi automaton over infinite words can be expressed using fairness constraints. This in turn captures the model checking problem for linear-time temporal logics [VW86,Kur93].

Weak monadic second-order logic of a single successor (WS1S). WS1S has a successor operation for constructing natural numbers, first-order quantification over natural numbers, and second-order quantification over finite sets of natural numbers. WS1S is a natural formalism for many applications, particularly for parametric systems. The logic can be used to capture interesting datatypes such as regular expressions, lists, queues, and arrays. There is a direct mapping between the logic and finite automata. A finite set X of natural numbers can be represented as a bit-string where a 1 in the i'th position indicates that i is a member of X. A formula with free set variables $X_1, \ldots, X_n$ is then a set of strings over B^n. The logical operations have automata theoretic counterparts so that negation is complementation, conjunction is the product of automata, and existential quantification is projection. The MONA library [EKM98] uses an ROBDD representation for the automaton corresponding to the formula.

3 Equality and Inequality

Equality introduces some of the most significant challenges in automated reasoning [HO80]. Many subareas of theorem proving are devoted to equality including rewriting, constraint solving, and unification. In this section we focus on ground decision procedures for equality. Many theorem proving systems are based around decision procedures for equality. The language now includes terms which are built from variables x, and applications $f(a_1, \ldots, a_n)$ of an n-ary function symbol f to n terms $a_1, \ldots, a_n$. The *ground* fragment can be seen as an extension of propositional logic where the propositional atoms are of the form

$a = b$, for terms a and b. The literals are now either equations $a = b$ or disequations $a \neq b$. The variables in a formula are taken to be universally quantified. The validity of a formula ϕ that is a propositional combination of equalities can be decided by first transforming $\neg\phi$ into disjunctive normal form $D_1 \vee \ldots \vee D_n$, and checking that each disjunct D_i, which is a conjunction of literals, is refutable. The refutation of a conjunction D_i of literals can be carried out by partitioning the terms in D_i into equivalence classes of terms with respect to the equalities in D_i. If for some disequation $a \neq b$ in D_i, a and b appear in the same equivalence class, then we have a contradiction and D_i has been refuted. The original claim ϕ is verified if each such disjunct D_i has been refuted.

If the function symbols are all uninterpreted, then congruence closure can be used to construct the equivalence classes corresponding to the conjunction of literals D_i. Let the set of subterms of D_i be $[\![D_i]\!]$. The initial partition P_0 is the set $\{\{c\} \mid c \in [\![D_i]\!]\}$. When an equality of the form $a = b$ from D_i is processed, it results in the merging of the equivalence classes corresponding to a and b. As a result of this merge, other equivalence classes might become mergeable. For example, one equivalence might contain $f(a_1, \ldots, a_n)$ while the other contains $f(b_1, \ldots, b_n)$, and each a_j is in the same equivalence class as the corresponding b_j. The merging of equivalence classes is performed until no further mergeable pairs of equivalence classes remain, and the partition P_1 is constructed. The equalities in D_i are successively processed and the resulting partition is returned as P_m. If for some disequality $a \neq b$, a and b are in the same equivalence class in P_m, then a contradiction is returned. Otherwise, the conjunction D_i is satisfiable.

Linear arithmetic. A large fraction of the subgoals that arise in verification condition generation, typechecking, array-bounds checking, and constraint solving involve linear arithmetic constraints [BW01]. Linear arithmetic equalities in n variables have the form $c_0 + c_1 * x_1 + \ldots + c_n * x_n = 0$, where the coefficients c_i range over the rationals, and the variables x_i range over the rationals or reals. It is easy to isolate a single variable, say x_1, as $x_1 = -c_0/c_1 - (c_2/c_1) * x_2 - \ldots - (c_n/c_1) * x_n$. This solved form for x_1 can then be substituted into the remaining linear equations thus eliminating the variable x_1. Gaussian elimination is based on the same idea where the set of linear equations is represented by $A * X = B$, and the matrix representation of the linear equations is transformed into row echelon form in order to solve for the variables.

Linear inequalities are of the form $c_0 + c_1 * x_1 + \ldots + c_n * x_n \ \# \ 0$, where $\#$ is either $<$, $\leq$, $>$, or $\geq$. Note that linear inequalities, unlike equalities, are closed under negation. Any linear equality can also be easily transformed into a pair of inequalities. As with linear equalities, linear inequalities can also be transformed into a form where a single variable is isolated. A pair of inequalities, $x \leq a$ and $x \geq b$ can be resolved to obtain $b \leq a$ thus eliminating x. This kind of Fourier-Motzkin elimination [DE73] can be used as a quantifier elimination procedure to decide the first-order theory of linear arithmetic by repeatedly reducing any quantified formula of the form $\exists x : P(x)$ where $P(x)$ is a conjunction of inequalities, into the form P', where x has been eliminated. By eliminating quantifiers

in an inside-out order while transforming universal quantification $\forall x : A$ into $\neg\exists x : \neg A$, we arrive at an equivalent variable-free formula that directly evaluates to true or false. Linear programming techniques like Simplex [Nel81] can also be used for solving linear arithmetic inequality constraints. Separation predicates are linear inequalities of the form $x - y \leq c$ or $x - y < c$ for some constant c, and these can be decided with graph-theoretic techniques [Sho81]. This simple class of linear inequalities is useful in model checking timed automata [ACD93].

Presburger arithmetic [Pre29] is the first-order theory of linear arithmetic over the integers. Solving constraints over the integers is harder than over the rationals and reals. Cooper [Coo72,Opp78] gives an efficient quantifier elimination algorithm for Presburger arithmetic. Once again, we need only consider quantifiers of the form $\exists x : P(x)$ where $P(x)$ is a conjunction of inequalities. We add divisibility assertions of the form $k|a$, where k is a positive integer. An inequality of the form $c_0 + c_1 * x_1 + \ldots + c_n * x_n \geq 0$ can be transformed to $c_1 * x_1 \geq -c_0 - c_2 * x_n - \ldots - c_n * x_n$, and similarly for other inequality relations. Since we are dealing with integers, a nonstrict inequality like $a \leq b$ can be transformed to $a < b + 1$. Having isolated all occurrences of x_1, we can compute the least common multiple α_1 of the coefficients corresponding to each occurrence of x_i. Now $P(x_1)$ is of the form $P'(\alpha_1 * x_1)$, and $\exists x_1 : P(x_1)$ can be replaced by $\exists x_1 : P'(x_1) \wedge \alpha_1 | x_1$. Here, $P'(x)$ is a conjunction of formulas of the forms: $x < a$, $x > b$, $k|x+d$, and $j \not| x+e$. Let $A = \{a \mid x < a \in P'(x)\}$, $B = \{b \mid x > b \in P'(x)\}$, $K = \{k \mid (k|x+d) \in P'(x)$, and $J = \{j \mid (j \not| x+e) \in P'(x)\}$. Let G be the least common multiple of $K \cup J$. If A is nonempty, then $\exists x : P'(x)$ can be transformed to $\bigvee_{a \in A} \exists x : a - G \leq x < a \wedge P'(x)$. The bounded existential quantification in the latter formula can easily be eliminated. Essentially, if m satisfies the constraints in $K \cup J$, then so does $m + r * G$ for any integer r. Hence, if $P'(m)$ holds for some m and A is nonempty, then there is an m in the interval $[a - G, a)$ for some $a \in A$ such that $P'(m)$ holds. Similarly, if B is nonempty, $\exists x : P'(x)$ can also be transformed to $\bigvee_{b \in B} \exists x : b < x \leq b + G \wedge P'(x)$. If both A and B are empty, then $\exists x : P'(x)$ is transformed to $\exists x : 0 < x \leq G \wedge P'(x)$. For example, the claim that x is an even integer can be expressed as $\exists u : 2 * u = x$ if we avoid the divisibility predicate. The quantifier elimination transformation above would convert this to $u' > x - 1 \wedge u' < x + 1 \wedge (2|u')$ which eventually yields $(x > x - 1 \wedge x < x + 1 \wedge 2|x) \vee (x + 1 > x - 1 \wedge x + 1 < x + 1 \wedge (2|x + 1))$. The latter formula easily simplifies to $(2|x)$. The claim that the sum of two even numbers is even then has the form $(\forall x : \forall y : 2|x \wedge 2|y \supset 2|(x + y))$. Converting universal quantification to existential quantification yields $\neg\exists x : \exists y : 2|x \wedge 2|y \wedge 2 \not|(x + y)$. Quantifier elimination yields $\neg\exists x : 0 < x \leq 2 \wedge \exists y : 0 < y \leq 2 \wedge (2|x) \wedge (2|y) \wedge (2 \not| x + y)$, which is clearly valid. The decidability of Presburger arithmetic can also be reduced to that of WS1S, and even though the latter theory has nonelementary complexity, this reduction using MONA works quite efficiently in practice [SKR98].

By the unsolvability of Hilbert's tenth problem, even the quantifier-free fragment of nonlinear arithmetic over the integers or rationals is undecidable. However, the first-order theory of nonlinear arithmetic over the reals and the complex

numbers is decidable. Tarski [Tar48] gave a decision procedure for this theory. Collins [Col75] gave an improved quantifier elimination procedure that is the basis for a popular package called QEPCAD [CH91]. These procedures have been successfully used in proving theorems in algebraic geometry. Buchberger's Gröbner basis method for testing membership in polynomial ideals has also been successful in computer algebra and geometry theorem proving [CG01,BW01].

Constraint solving and quantifier elimination methods in linear and nonlinear arithmetic over integers, reals, and rationals, are central to a large number of applications of theorem proving that involve numeric constraints.

4 The Combination Problem

The application of decision procedures for individual theories is constrained by the fact that few natural problems fall exactly within a single theory. Many of the proof obligations that arise out of extended typechecking or verification condition generation involve arithmetic equalities and inequalities, tuples, arrays, datatypes, and uninterpreted function symbols. There are two basic techniques for constructing decision procedures for checking the satisfiability of conjunctions of literals in combinations of disjoint theories: the Nelson–Oppen method [NO79,TH96] and the Shostak method [Sho84].

Nelson and Oppen's Method. The Nelson–Oppen method combines decision procedures for disjoint theories by using variable abstraction to purify a formula containing operations from a union of theories, so that the formula can then be partitioned into subgoals that can be handled by the individual decision procedures. Let B represent the formula whose satisfiability is being checked in the union of disjoint theories θ_1 and θ_2. First variable abstraction is used to convert B into $B' \wedge V$, where V contains equalities of the form $x = t$, where x is a fresh variable and t contains function symbols exclusively from θ_1 or from θ_2, and B' contains x renaming t. In particular, if $V[B']$ is the result of replacing each occurrence of x in B' by the corresponding t for each $x = t$ in V, then B must the result of repeatedly applying V to B' and eliminating all the newly introduced variables. Next, $V \wedge B'$ can be partitioned as $B_1 \wedge B_2$, where each B_i only contains function symbols from the theory θ_i. Let X be the free variables that are shared between B_1 and B_2. Guess a partition $X_1, \ldots, X_m$ on the variables in X. Let E be an arrangement corresponding to this partition so that E contains $x = y$ for each pair of distinct variables x, y in some X_j, and $u \neq v$ for each pair of variables u, v, such that $u \in X_j, v \in X_k$ for $j \neq k$. Check if $E \wedge B_1$ is satisfiable in θ_1 and $E \wedge B_2$ is satisfiable in θ_2. If that is the case, then B is satisfiable in $\theta_1 \cup \theta_2$, provided θ_1 and θ_2 are *stably infinite*. A theory θ is stably infinite if whenever a formula is θ-satisfiable (satisfiable in a θ-model), it is θ-satisfiable in an infinite model.

Shostak's Method. The Nelson–Oppen combination is a way of combining black box decision procedures. Shostak's method is an optimization of the Nelson–

Oppen combination for a restricted class of equational theories. A theory θ is said to be canonizable if there is a canonizer σ such that the equality $a = b$ is valid in θ iff $\sigma(a) \equiv \sigma(b)$. A theory θ is said to be solvable if there is an operation *solve* such that $solve(a = b)$ returns a set S of equalities $\{x_1 = t_1, \ldots, x_n = t_n\}$ equivalent in some sense to $a = b$, where each x_i occurs in $a = b$ but not in t_j for $1 \leq i, j \leq n$. A Shostak theory is one that is canonizable and solvable. Shostak's combination method can be used to combine one or more Shostak theories with the theory of equality over uninterpreted terms. The method essentially maintains a set S of solutions $S_0, \ldots, S_N$, where each set S_i contains equalities of the form $x = t$ for some term t in θ_i. The theory θ_0 is used for the uninterpreted function symbols. Two variables x and y are said to be merged in S_i if $x = t$ and $y = t$ are both in S_i. It is possible to define a global canonical form $S[\![a]\!]$ for a term a with respect to the solution state S using the individual canonizers σ_i.

Shostak's original algorithm [Sho84] and its proof were both incorrect. The algorithm, as corrected by the author and Harald Ruess [RS01,SR02], checks the validity of a sequent $T \vdash c = d$. It does this by processing each equality $a = b$ into its solved form. If S is the current solution state, then an unprocessed equality $a = b$ in T is processed by first transforming it to $a' = b'$, where $a' = S[\![a]\!]$ and $b' = S[\![b]\!]$. The equality $a' = b'$ is variable abstracted and the variable abstraction equalities $x = t$ are added to the solution S_i, where t is a term in the theory θ_i. The algorithm then repeatedly reconciles the solutions S_i so that whenever two variables x and y are merged in S_i but not in S_j, for $i \neq j$, then they are merged in S_j by solving $t_x = t_y$ in θ_j, for $x = t_x$ and $y = t_y$ in S_j, and composing the solution with S_j to obtain a new solution set S_j. When all the input equalities from T have been processed and we have the resulting solution state S, we check if $S[\![c]\!] = S[\![d]\!]$. A conjunction of literals $\bigwedge_{i=1}^{m} a_i = b_i \wedge \bigwedge_{j=1}^{n} c_j \neq d_j$ is satisfiable iff $S \neq \bot$ and $S[\![c_j]\!] \not\equiv S[\![d_j]\!]$, for each j, $1 \leq j \leq n$, where $S = process(\{a_1 = b_1, \ldots, a_m = b_m\})$.

Ground Satisfiability. The Nelson–Oppen and Shostak decision procedures check the satisfiability of conjunctions of literals drawn from a combination of theories. These procedures can be extended to handle propositional combinations of atomic formulas by transforming these formulas to disjunctive normal form. This method can be inefficient when the propositional case analysis involved is heavy. It is usually more efficient to combine a SAT solver with a ground decision procedure [BDS02,dMRS02]. There are various ways in which such a combination can be executed. Let ϕ be the formula whose satisfiability is being checked. Let L be an injective map from fresh propositional variables to the atomic subformulas of ϕ such that $L^{-1}[\phi]$ is a propositional formula. We can use a SAT solver to check that $L^{-1}[\phi]$ is satisfiable, but the resulting truth assignment, say $l_1 \wedge \ldots \wedge l_n$, might be spurious, that is $L[l_1 \wedge \ldots \wedge l_n]$ might not be ground-satisfiable. If that is the case, we can repeat the search with the added lemma $(\neg l_1 \vee \ldots \vee \neg l_n)$ and invoke the SAT solver on $(\neg l_1 \vee \ldots \vee \neg l_n) \wedge L^{-1}[\phi]$. This ensures that the next satisfying assignment returned is different from the previous assignment that was found to be ground-unsatisfiable. The lemma that is added can be minimized to find the minimal unsatisfiable set of literals l_i.

This means that the lemma that is added is smaller, and the pruning of spurious assignments is more effective. The ground decision procedure can be also be used to precompute a set Λ of lemmas (clauses) of the form $l_1 \vee \ldots \vee l_n$, where $\neg L[l_1] \wedge \ldots \neg L[l_n]$ is unsatisfiable according to the ground decision procedures. The SAT solver can then be reinvoked with $\Lambda \wedge L^{-1}[\phi]$.

A tighter integration of SAT solvers and ground decision procedures would allow the decision procedures to check the consistency of the case analysis during an application of splitting in the SAT solver and avoid cases that are ground-unsatisfiable. Through a tighter integration, it would also be possible to resume the SAT solver with the added conflict information without starting the SAT solving process from scratch. We address the challenge of integrating inference procedures below.

Applications. Ground decision procedures, ground satisfiability, and quantifier elimination have many applications.

Symbolic Execution: Given a transition system, symbolic execution is the process of computing preconditions or postconditions of the transition system with respect to an assertion. For example, the strongest postcondition of an assertion p with respect to a transition N is the assertion $\lambda s : \exists s_0 : p(s_0) \wedge N(s_0, s)$. For certain choices of p and N, this assertion can be computed by means of a quantifier elimination. This is useful in analyzing timed and hybrid systems [ACH+95].

Infinite-State Bounded Model Checking: Bounded model checking checks for the existence of counterexamples of length upto a bound k for a given temporal property. With respect to certain temporal properties, it is possible to reduce the bounded model checking problem for such systems to a ground satisfiability problem [dMRS02].

Abstraction and Model Checking: The early work on abstraction in the context of model checking was on reducing finite-state systems to smaller finite-state systems, i.e., systems with fewer possible states [Kur93,CGL92,LGS+95]. Graf and Saïdi [GS97] were the first to consider the use of a theorem prover for reducing (possibly) infinite-state systems to finite-state (hence, model-checkable) form. Their technique of *predicate abstraction* constructs an abstract counterpart of a concrete transition system where the truth values of certain predicates over the concrete state space are simulated by boolean variables. Data abstraction replaces a variable over an infinite state space by one over a finite domain. Predicate and data abstraction based on theorem proving are widely used [BLO98b,CU98,DDP99,SS99,BBLS00,CDH+00,TK02,HJMS02,FQ02]. The finite-state abstraction can exhibit spurious counterexamples that are not reproducible on the concrete system. Ground decision procedures are also useful here for detecting spurious counterexamples and suggesting refinements to the abstraction predicates [BLO98a,SS99,DD01].

Software Engineering: Ground decision procedures are central to a number of analysis tools for better engineered software including array-bounds check-

ing, extended static checking [DLNS98], typechecking [SO99], and static analysis [BMMR01,Pug92].

5 Challenges

We have enumerated some of the progress in developing, integrating, and deploying various inference procedures. A great many challenges remain. We discuss a few of these below.

The Complexity Challenge. Many decision procedures are of exponential, super-exponential, or non-elementary complexity. However, this complexity often does not manifest itself on practical examples. Modern SAT solvers can solve very large practical problems, but they can also run aground on small instances of simple challenges like the propositional pigeonhole principle. MONA deals with a logic that is known to have a non-elementary lower bound, yet it performs quite well in practice. The challenge here is to understand the ways in which one can overcome complexity bounds on the problems that arise in practice through heuristic or algorithmic means.

The Theory Challenge. Inference procedures are hard to build, extend, and maintain. The past experience has been that good theory leads to simpler decision procedures with greater efficiency. A well-developed theory can also help devise uniform design patterns for entire classes of decision procedures. Such design patterns can contribute to both the efficiency and modularity of these procedures. Methods derived by specializing general-purpose methods like resolution and rewriting can also simplify the construction of decision procedures.

The Modularity Challenge. As we have already noted, inference procedures need rich programmer interfaces (APIs) [BM86,FORS01]. Boyer and Moore [BM86] write:

> *... the black box nature of the decision procedure is frequently destroyed by the need to integrate it. The integration forces into the theorem prover much knowledge of the inner workings of the procedure and forces into the procedure many features that are unnecessary when the problem is considered in isolation.*

For example, a ground decision procedure can be used in an online manner so that atomic formulas are added to a context incrementally, and claims are tested against the context. The API should include operations for asserting and retracting information, testing claims, and for creating, deleting, and browsing contexts. The decision procedures might need to exchange information with other inference procedures such as a rewriter, typechecker, or an external constraint solver. We already saw how the desired interaction between ground decision procedures and SAT solvers was such that neither of these could be treated as a black box procedure.

The modularity challenge is a significant one. Butler Lampson has argued that software components have always failed at low levels of granularity (see `http://research.microsoft.com/users/blampson/Slides/ReusableComponentsAbstract.htm`). He says that successful software components are those at the level of a database, a compiler, or a theorem prover, but not decision procedures, constraint solvers, or unification procedures. For interoperation between inference components, we also need compatible logics, languages, and term and proof representations.

The Integration Challenge. The availability of good inference components is a prerequisite for integration, but we also need to find effective ways of combining these components in complementary ways. The combination of decision procedures with model checking in predicate and data abstraction is a case where such a complementary integration is remarkably effective. Other such examples include the combination of unification/matching procedures and constraint solving, and typechecking and ground decision procedures.

The Verification Challenge. How do we know that our inference procedures are sound? This question is often asked by those who wish to apply inference procedures in contexts where a high level of manifest assurance is required. This question has been addressed in a number of ways. The LCF approach [GMW79] requires inference procedures to be constructed as tactics that generate a fully expanded proof in terms of low level inferences when applied. Proof objects have also been widely used as a way of validating inference procedures and securing mobile code [Nec97]. Reflection [Wey80,BM81] is a way of reasoning about the metatheory of a theory within the theory itself. The difficult tradeoff with reflection is that the theory has to be simple in order to be reasoned about, but rich enough to reason with. The verification of decision procedures is actually well within the realm of feasible, and recently, there have been several successful attempts in this direction [Thé98,FS02].

6 Conclusions

We have argued for a reappraisal of Hao Wang's programme [Wan60b,Wan60a] of inferential analysis as a paradigm for automated reasoning. The key element of this paradigm is the use of problem-driven combinations of sophisticated and efficient low-level decision procedures. Such an approach runs counter to the traditional thinking in automated reasoning which is centered around uniform proof search procedures. Similar ideas are also central to the automated reasoning schools of Bledsoe [Ble77] and Boyer and Moore [BM79,BM86].

The active use of decision procedures in automated reasoning began with the *west-coast* theorem proving approach pioneered by Boyer and Moore [BM79], Shostak [SSMS82], and Nelson and Oppen [LGvH$^+$79,NO79]. The PVS system is in this tradition [ORS92,Sha01], as are STeP [MT96], SIMPLIFY [DLNS98], and SVC [BDS00].

In recent years there has been a flurry of interest in the development of verification tools that rely quite heavily on sophisticated decision procedures. The quality and efficiency of many of these decision procedures is impressive. The underlying theory is also advancing rapidly [Bjø99,Tiw00]. Such theoretical advances will make it easier to construct correct decision procedures and integrate them more easily with other inference mechanisms. Contrary to the impression that decision procedures are black boxes, they need rich interfaces [BM86,FORS01,GNTV02] in order to be deployed most efficiently. The theory, construction, integration, verification, and deployment of inference procedures is likely to be a fertile source of challenges for automated reasoning in mathematically rich domains.

References

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 6 February 1995.

[BBLS00] Kai Baukus, Saddek Bensalem, Yassine Lakhnech, and Karsten Stahl. Abstracting WS1S systems to verify parameterized networks. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, number 1785 in Lecture Notes in Computer Science, pages 188–203, Berlin, Germany, March 2000. Springer-Verlag.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BDS00] Clark W. Barrett, David L. Dill, and Aaron Stump. A framework for cooperating decision procedures. In David McAllester, editor, *Automated Deduction—CADE-17*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 79–98, Pittsburgh, PA, June 2000. Springer-Verlag.

[BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Computer-Aided Verification, CAV '02*, Lecture Notes in Computer Science. Springer-Verlag, July 2002.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Robinson and Voronkov [RV01], pages 19–99.

[BGG97] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.

[Bjø99] Nikolaj Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Stanford University, 1999.

[Ble77] W. W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9:1–36, 1977.

[BLO98a] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [HV98], pages 319–331.

[BLO98b] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In Hu and Vardi [HV98], pages 505–510.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[BM81] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[BM86] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence*, volume 11. Oxford University Press, 1986.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation, 2001*, pages 203–313. ACM Press, 2001.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BW01] Alexander Bockmayr and Volker Weispfenning. Solving numerical constraints. In Robinson and Voronkov [RV01], pages 751–742.

[CDH+00] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.

[CG01] Shang-Ching Chou and Xiao-Shan Gao. Automated reasoning in geometry. In Robinson and Voronkov [RV01], pages 707–749.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[CGP99] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CH91] G. E. Collins and H. Hong. Partial cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 12(3):299–328, 1991.

[Col75] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Second GI Conference on Automata Theory and Formal Languages*, number 33 in Lecture Notes in Computer Science, pages 134–183, Berlin, 1975. Springer-Verlag.

[Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence 7*, pages 91–99. Edinburgh University Press, 1972.

[CU98] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decidion procedures. In Hu and Vardi [HV98], pages 293–304.

[Dav57] M. Davis. A computer program for Presburger's algorithm. In *Summaries of Talks Presented at the Summer Institute for Symbolic Logic*, 1957. Reprinted in Siekmann and Wrightson [SW83], pages 41–48.

[Dav83] M. Davis. The prehistory and early history of automated deduction. In Siekmann and Wrightson [SW83], pages 1–28.

[DD01] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Annual IEEE Symposium on Logic in Computer Science01*, pages 51–60. The Institute of Electrical and Electronics Engineers, 2001.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171, Trento, Italy, July 1999. Springer-Verlag.

[DE73] George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962. Reprinted in Siekmann and Wrightson [SW83], pages 267–270, 1983.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[dMRS02] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *International Conference on Automated Deduction (CADE'02)*, Lecture Notes in Computer Science, Copenhagen, Denmark, July 2002. Springer-Verlag.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7(3):201–215, 1960.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Möller. Mona 1.x: New techniques for WS1S and WS2S. In Hu and Vardi [HV98], pages 516–520.

[FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification, CAV '2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249, Paris, France, July 2001. Springer-Verlag.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *ACM Symposium on Principles of Programming Languages02*, pages 191–202. Association for Computing Machinery, January 2002.

[FS02] Jonathan Ford and Natarajan Shankar. Formal verification of a combination decision procedure. In A. Voronkov, editor, *Proceedings of CADE-19*, Berlin, Germany, 2002. Springer-Verlag.

[Gil60] P. C. Gilmore. A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development*, 4:28–35, 1960. Reprinted in Siekmann and Wrightson [SW83], pages 151–161, 1983.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[GNTV02] Enrico Giunchiglia, Massimo Narizzano, Armando Tacchella, and Moshe Y. Vardi. Towards an efficient library for SAT: a manifesto. To appear, 2002.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *ACM Symposium on Principles of Programming Languages02*, pages 58–70. Association for Computing Machinery, January 2002.

[HO80] G. Huet and D. C. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, ny, 1980.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*, volume 3 of *Advances in Formal Methods*. Kluwer, 2000.

[Kur93] R.P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1993.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.

[LGvH+79] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. CSD Report STAN-CS-79-731, Stanford University, Stanford, CA, March 1979.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[Min63] Marvin Minsky. Steps toward artificial intelligence. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill Book Company, New York, 1963.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[MSS99] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[MT96] Zohar Manna and The STeP Group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Nec97] George C. Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997. Association for Computing Machinery.

[Nel81] G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, Palo Alto, Ca., 1981.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NSS57] A. Newell, J. C. Shaw, and H. A. Simon. Empirical explorations with the logic theory machine: A case study in heuristics. In *Proc. West. Joint Comp. Conf.*, pages 218–239, 1957. Reprinted in Siekmann and Wrightson [SW83], pages 49–73, 1983.

[Opp78] Derek C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16:323–332, 1978.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pos21] E. L. Post. Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, 43:163–185, 1921. Reprinted in [vH67, pages 264–283].

[Pra60] D. Prawitz. An improved proof procedure. *Theoria*, 26:102–139, 1960. Reprinted in Siekmann and Wrightson [SW83], pages 162–201, 1983.

[Pre29] M. Presburger. Uber die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. *Compte Rendus du congrés Mathématiciens des Pays Slaves*, pages 92–101, 1929.

[Pug92] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[Rab78] Michael O. Rabin. Decidable theories. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter C8, pages 595–629. North-Holland, Amsterdam, Holland, 1978.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965. Reprinted in Siekmann and Wrightson [SW83], pages 397–415.

[RS01] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, Boston, MA, July 2001. IEEE Computer Society.

[RV01] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science, 2001.

[Sha01] Natarajan Shankar. Using decision procedures with a higher-order logic. In *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 5–26, Edinburgh, Scotland, September 2001. Springer-Verlag. Available at `ftp://ftp.csl.sri.com/pub/users/shankar/tpholls2001.ps.gz`.

[Sho81] Robert E. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[SKR98] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of Presburger engines for EFSM reachability. In Hu and Vardi [HV98], pages 280–292.

[SO99] Natarajan Shankar and Sam Owre. Principles and pragmatics of subtyping in PVS. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT '99*, volume 1827 of *Lecture Notes in Computer Science*, pages 37–52, Toulouse, France, September 1999. Springer-Verlag.

[SR02] N. Shankar and H. Rueß. Combining Shostak theories. In *International Conference on Rewriting Techniques and Applications (RTA '02)*, Lecture Notes in Computer Science. Springer-Verlag, July 2002. Invited Paper.

[SS99] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In *Computer-Aided Verification, CAV '99*, Trento, Italy, July 1999.

[SS00] Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in Systems Design*, 16(1):23–58, January 2000.

[SSMS82] R. E. Shostak, R. Schwartz, and P. M. Melliar-Smith. STP: A mechanized logic for specification and verification. In D. Loveland, editor, *6th International Conference on Automated Deduction (CADE)*, volume 138 of *Lecture Notes in Computer Science*, New York, NY, 1982. Springer-Verlag.

[SW83] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning: Classical Papers on Computational Logic, Volumes 1 & 2*. Springer-Verlag, 1983.

[Tar48] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1948.

[TH96] Cesare Tinelli and Mehdi Harandi. A new correctness proof of the Nelson-Oppen combination procedure. In Frans Baader and Klaus U. Schulz, editors, *Frontiers of Combining Systems: First International Workshop*, volume 3 of *Applied Logic Series*, pages 103–119, Munich, Germany, March 1996. Kluwer.

[Thé98] Laurent Théry. A certified version of Buchberger's algorithm. In H. Kirchner and C. Kirchner, editors, *Proceedings of CADE-15*, number 1421 in Lecture Notes in Artificial Intelligence, pages 349–364, Berlin, Germany, July 1998. Springer-Verlag.

[Tiw00] Ashish Tiwari. *Decision Procedures in Automated Deduction*. PhD thesis, State University of New York at Stony Brook, 2000.

[TK02] Ashish Tiwari and Gaurav Khanna. Series of abstractions for hybrid automata. In C.J. Tomlin and M.R. Greenstreet, editors, *Hybrid Systems: Computation and Control, 5th International Workshop, HSCC 2002*, volume 2289 of *Lecture Notes in Computer Science*, pages 465–478, Stanford, CA, March 2002. Springer-Verlag.

[vH67] J. van Heijenoort, editor. *From Frege to Gödel: A Sourcebook of Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1967.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science*, pages 332–344. IEEE Computer Society Press, 1986.

[Wan60a] H. Wang. Proving theorems by pattern recognition — I. *Communications of the ACM*, 3(4):220–234, 1960. Reprinted in Siekmann and Wrightson [SW83], pages 229–243, 1983.

[Wan60b] Hao Wang. Toward mechanical mathematics. *IBM Journal*, 4:2–22, 1960.

[Wan63] H. Wang. Mechanical mathematics and inferential analysis. In P. Braffort and D. Hershberg, editors, *Computer Programming and Formal Systems*. North-Holland, 1963.

[Wey80] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13(1 and 2):133–170, April 1980.

[Zha97] Hantao Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction*, pages 272–275, 1997.

Deconstructing Shostak*

Appears in Proc. of IEEE LICS 2001 ©IEEE Press

Harald Rueß and Natarajan Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{ruess,shankar}@csl.sri.com
Phone: (650)859-5272; Fax: (650)859-2844

Abstract

Decision procedures for equality in a combination of theories are at the core of a number of verification systems. Shostak's decision procedure for equality in the combination of solvable and canonizable theories has been around for nearly two decades. Variations of this decision procedure have been implemented in a number of systems including STP, EHDM, *PVS, STeP, and SVC. The algorithm is quite subtle and a correctness argument for it has remained elusive. Shostak's algorithm and all previously published variants of it yield incomplete decision procedures. We describe a variant of Shostak's algorithm along with proofs of termination, soundness, and completeness.*

1 Introduction

In 1984, Shostak [Sho84] published a decision procedure for the quantifier-free theory of equality over uninterpreted functions combined with other theories that are canonizable and solvable. Such algorithms decide statements of the form $T \vdash a = b$, where T is a collection of equalities, and T, a, and b contain a mixture of interpreted and uninterpreted function symbols. This class of statements includes a large fraction of the proof obligations that arise in verification including those involving extended typechecking, verification conditions generated from Hoare triples, and inductive theorem proving. Shostak's procedure is at the core of several verification systems including STP [SSMS82], EHDM [EHD93], PVS [ORS92], STeP [MT96, Bjø99], and SVC [BDL96]. The soundness of Shostak's algorithm is reasonably straightforward, but its completeness has steadfastly resisted proof. The proof given by Shostak [Sho84] is seriously flawed. Despite its significance and popularity, Shostak's original algorithm and its subsequent variations [CLS96, BDL96, Bjø99] are all incomplete and potentially nonterminating. We explain the ideas underlying Shostak's decision procedure by presenting a correct version of the algorithm along with rigorous proofs for its correctness.

If the terms in a conjecture of the form $T \vdash a = b$ are constructed solely from variables and uninterpreted function symbols, then congruence closure [NO80, Sho78, DST80, CLS96, Kap97, BRRT99] can be used to partition the subterms into equivalence classes respecting T and congruence. For example, when congruence closure is applied to

$$f^3(x) = f(x) \vdash f^5(x) = f(x),$$

the equivalence classes generated by the antecedent equality are $\{x\}, \{f(x), f^3(x), f^5(x)\}$, and $\{f^2(x), f^4(x)\}$. This partition clearly validates the conclusion $f^5(x) = f(x)$.

In practice, a conjecture $T \vdash a = b$ usually contains a mixture of uninterpreted and interpreted function symbols. Semantically, uninterpreted functions are unconstrained, whereas interpreted function are constrained by a *theory*, i.e., a closure condition with respect to consequence on a set of equalities. An example of such an assertion is

$$f(x-1)-1 = x+1,\ f(y)+1 = y-1,\ y+1 = x \vdash \mathit{false},$$

where $+$, $-$, and the numerals are from the theory of linear arithmetic, *false* is an abbreviation for $0 = 1$, and f is an uninterpreted function symbol. The contradiction here cannot be derived solely by congruence closure or linear arithmetic. Linear arithmetic is used to show that $x - 1 = y$ so that $f(x-1) = f(y)$ follows by congruence. Linear arithmetic can then be used to show that $x + 2 = y - 2$ which contradicts $y + 1 = x$.

*This work was supported by SRI International, and by NSF Grant CCR-0082560, DARPA/AFRL Contract F33615-00-C-3043, and NASA Contract NAS1-0079.

Nelson and Oppen [NO79] showed how decision procedures for disjoint equational theories could be combined. Since linear arithmetic and uninterpreted equality are disjoint, this method can be applied to the above example. First, *variable abstraction* is used to obtain a theory-wise partition of the *term universe*, i.e., the subterms of T, a, and b, in a conjecture $T \vdash a = b$. The uninterpreted equality theory Q then consists of the terms $\{f(u), f(y), w, z\}$ and the equalities $\{w = f(u), z = f(y)\}$, and the linear arithmetic theory L consists of the terms $\{u, x, y, x-1, w-1, x+1, z+1, y-1, y+1\}$ and the equalities $\{u = x-1, w-1 = x+1, z+1 = y-1, y+1 = x\}$. The key observation is that once the terms and equalities have been partitioned using variable abstraction, the two theories L and Q need exchange only equalities between variables. Thus, linear arithmetic can be used to derive the equality $u = y$, from which congruence closure derives $w = z$, and the contradiction then follows from linear arithmetic. Since the term universe is fixed in advance, there are only a bounded number of equalities between variables so that the propagation of information between the decision procedures must ultimately converge.

The Nelson-Oppen combination procedure has some disadvantages. The individual decision procedures must carry out their own equality propagation and the communication of equalities between decision procedures can be expensive. The number of equalities is quadratic in the size of the term universe, and each closure operation can itself be linear in the size of the term universe.

Shostak's algorithm tries to gain efficiency by maintaining and propagating equalities within a single congruence closure data structure. Equalities involving interpreted symbols contain more information than uninterpreted equalities. For example, the equality $y+1 = x$ cannot be processed by merely placing $y+1$ and x in the same equivalence class. This equality also implies that $y = x-1$, $y - x = -1$, $x - y = 1$, $y+3 = x+2$, and so on. In order to avoid processing all these variations on the given equality, Shostak restricts his attention to *solvable* theories where an equality of the form $y+1 = x$ can be solved for x to yield the solution $x = y+1$. If the theories considered are also *canonizable*, then there is a canonizer σ such that whenever an equality $a = b$ is valid, then $\sigma(a) \equiv \sigma(b)$, where $\equiv$ represents syntactic equality. A canonizer for linear arithmetic can be defined to place terms into an ordered sum-of-monomials form. Once a solved form such as $x = y+1$ has been obtained, all the other consequences $a = b$ of this equality can be obtained by $\sigma(a') = \sigma(b')$ where a' and b' are the results of substituting the solution for x into a and b, respectively. For example, substituting the solution into $y = x-1$ yields $y = y+1-1$, and the subsequent canonization step yields $y = y$.

The notion of a solvable and canonizable theory is extended to equalities involving a mix of interpreted and uninterpreted symbols by treating uninterpreted terms as variables. For the conjecture,

$$f(x-1)-1 = x+1,\ f(y)+1 = y-1,\ y+1 = x \vdash \mathit{false},$$

Shostak's algorithm would solve the equality $f(x-1)-1 = x+1$ as $f(x-1) = x+2$, the equality $f(y)+1 = y-1$ as $f(y) = y-2$, and $y+1 = x$ as $x = y+1$. Now, $f(x-1)$ and $f(y)$ are congruent because the canonical form for $x-1$ obtained after substituting the solution $x = y+1$ is y. By congruence closure, the equivalence classes of $f(x-1)$ and $f(y)$ have to be merged. In Shostak's original algorithm the current representatives of these equivalence classes, namely $x+2$ and $y-2$ are merged. The resulting equality $x+2 = y-2$ is first solved to yield $x = y-4$. This is incorrect because we already have a solution for x as $x = y+1$ and x should therefore have been eliminated. The new solution $x = y-4$ contradicts the earlier one, but this contradiction goes undetected by Shostak's algorithm. This example can be easily adapted to show nontermination. Consider

$$f(v) = v, f(u) = u-1, u = v \vdash \mathit{false}.$$

The merging of u and v here leads to the detection of the congruence between $f(u)$ and $f(v)$. This leads to solving of $u-1 = v$ as $u = v+1$. Now, the algorithm merges v and $v+1$. Since v occurs in $v+1$, this causes $v+1$ to be merged with $v+2$, and so on.

An earlier paper by Cyrluk, Lincoln, and Shankar [CLS96] gave an explanation (with minor corrections) of Shostak's algorithm for congruence closure and its extension to interpreted theories. Though proofs of correctness for the combination algorithm are briefly sketched, the algorithm presented there is both incomplete and nonterminating. Other published variants of Shostak's algorithm used in SVC [BDL96] and STeP [Bjø99] inherit these problems.

In this paper, we present an algorithm that fixes the incompleteness and nontermination in earlier versions of Shostak's algorithms. In the above example, the incompleteness is fixed by substituting the solution for x into the terms representing the different equivalence classes. Thus, when $f(x-1)$ and $f(y)$ are detected to be congruent, their equivalence classes are represented by $y+3$ and $y-2$, respectively. The resulting equality $y+3 = y-2$ easily yields a contradiction. The nontermination is fixed by ensuring that no new mergeable

terms, such as $v+2$, are created during the processing of an axiom in T. Our algorithm is presented as a system of transformations on a set of equalities in order to capture the key insights underlying its correctness. We outline rigorous proofs for the termination, soundness, and completeness of this procedure. The algorithm as presented here emphasizes logical clarity over efficiency, but with suitable optimizations and data structures, it can serve as the basis for an efficient implementation. SRI's ICS (Integrated Canonizer/Solver) decision procedure package [FORS01] is directly based on the algorithm studied here.

Section 2 introduces the theory of equality, which is augmented in Section 3 with function symbols from a canonizable and solvable theory. Section 3 also introduces the basic building blocks for the decision procedure. The algorithm itself is described in Section 4 along with some example hand-simulations. The proofs of termination, soundness, and completeness are outlined in Section 5.

2 Background

With respect to a *signature* consisting of a set of function symbols F and a set of variables V, a term is either a variable x from V or an application $f(a_1, \ldots, a_n)$ of an n-ary function symbol f from F to n terms $a_1, \ldots, a_n$, where $0 \leq n$. The metavariable conventions are that u, v, x, y, and z range over variables, and a, b, c, d, and e range over terms. The metavariables R, S, and T, range over sets of equalities. The metatheoretic assertion $a \equiv b$ indicates that a and b are syntactically identical terms. Let $vars(a)$, $vars(a = b)$, and $vars(T)$ return the variables occurring in a term a, an equality $a = b$, and a set of equalities T, respectively. The operation $[\![a]\!]$ is defined to return the set of all subterms of a.

Some of the function symbols are *interpreted*, i.e., they have a specific interpretation in some given theory τ, while the remaining function symbols are uninterpreted, i.e., can be assigned arbitrary interpretations. A term $f(a_1, \ldots, a_n)$ is interpreted (uninterpreted) if f is interpreted (uninterpreted). A term e is *non-interpreted* if it is either a variable or an uninterpreted term. We say that a term a *occurs interpreted* in a term e if there is an occurrence of a in e that is not properly within an uninterpreted subterm of e. Likewise, a *occurs uninterpreted* in e if a is a proper subterm of an uninterpreted subterm of e. $solvables(a)$ denotes the set of outermost non-interpreted subterms of a, i.e., those that do not occur uninterpreted in a.

$$\begin{aligned} solvables(f(a_1, \ldots, a_n)) &= \bigcup_i solvables(a_i), \\ &\quad \text{if } f \text{ is interpreted} \\ solvables(a) &= \{a\}, \text{ otherwise} \end{aligned}$$

The theory of equality deals with sequents of the form $T \vdash a = b$. We will insist that these sequents be such that $vars(a = b) \subseteq vars(T)$. The proof theory for equality is given by the following inference rules.

1. Axiom: $\dfrac{}{T \vdash a = b}$, for $a = b \in T$.
2. Reflexivity: $\dfrac{}{T \vdash a = a}$.
3. Symmetry: $\dfrac{T \vdash a = b}{T \vdash b = a}$.
4. Transitivity: $\dfrac{T \vdash a = b \qquad T \vdash b = c}{T \vdash a = c}$.
5. Congruence: $\dfrac{T \vdash a_1 = b_1 \quad \ldots \quad T \vdash a_n = b_n}{T \vdash f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)}$.

The semantics for terms is given by a model M over a domain D and an assignment ρ for the variables so that $M[\![x]\!]_\rho = \rho(x)$ and $M[\![f(a_1, \ldots, a_n)]\!]_\rho = M(f)(M[\![a_1]\!]_\rho, \ldots, M[\![a_n]\!]_\rho)$, and $M[\![a]\!]_\rho \in D$ for all a. We say that $M, \rho \models a = b$ iff $M[\![a]\!]_\rho = M[\![b]\!]_\rho$, and $M \models a = b$ iff $M, \rho \models a = b$ for all assignments ρ over $vars(a = b)$. We write $M, \rho \models S$ when $\forall a, b : a = b \in S \supset M, \rho \models a = b$, and $M, \rho \models T \vdash a = b$ when $(M, \rho \models T) \supset (M, \rho \models a = b)$.

3 Canonizable and Solvable Theories

Shostak's algorithm goes beyond congruence closure by deciding equality in the presence of function symbols that are *interpreted* in a theory τ [Sho84, CLS96]. The algorithm is targeted at canonizable and solvable theories, i.e., theories that are equipped with solvers and canonizers as outlined below. We write $\models_\tau a = b$ to indicate that $a = b$ is valid in theory τ. The canonizer and solver are first described for pure τ-terms, i.e., without any uninterpreted function symbols, and then extended to uninterpreted terms by regarding these as variables.

Definition 3.1 *A theory τ is* canonizable *if there is a canonizer σ such that*

1. $\models_\tau a = b$ *iff* $\sigma(a) \equiv \sigma(b)$.
2. $\sigma(x) \equiv x$.
3. $vars(\sigma(a)) \subseteq vars(a)$.
4. $\sigma(\sigma(a)) \equiv \sigma(a)$.
5. *If* $\sigma(a) \equiv f(b_1, \ldots, b_n)$, *then* $\sigma(b_i) \equiv b_i$ *for* $1 \leq i \leq n$.

For example, a canonizer σ for the theory of linear arithmetic can be defined to transform expressions into an ordered-sum-of-monomials normal form. A term a is said to be *canonical* if $\sigma(a) \equiv a$.

Definition 3.2 *A model M is a* σ-model *if $M \models a = \sigma(a)$ for any term a, and $M \not\models a = b$ for distinct canonical, variable-free terms a and b.*

Definition 3.3 *A set of equalities S and $a = b$ are* σ-equivalent *iff for all σ-models M and assignments ρ over the variables in a and b, $M, \rho \models a = b$ iff there is an assignment ρ' extending ρ, over the variables in S, a, and b, such that $M, \rho' \models S$.*

Definition 3.4 *A canonizable theory is* solvable *if there is an operation solve such that $solve(a = b) = \bot$ if $a = b$ is unsatisfiable in any σ-model, or $S = solve(a = b)$ for a set of equalities S such that*

1. *S is a set of n equalities of the form $x_i = e_i$ for $0 \leq n$ where for each i, $0 < i \leq n$,*
   (a) $x_i \in vars(a = b)$.
   (b) $x_i \notin vars(e_j)$, *for* j, $0 < j \leq n$.
   (c) $x_i \not\equiv x_j$, *for* $i \neq j$ *and* $0 < j \leq n$.
   (d) $\sigma(e_i) \equiv e_i$.
2. *S and $a = b$ are σ-equivalent.*

A solver for linear arithmetic, for example, takes an equation of the form

$$c + a_1x_1 + \ldots + a_nx_n = d + b_1x_1 + \ldots + b_nx_n,$$

where $a_1 \neq b_1$, and returns

$$\begin{array}{rcl} x_1 = \sigma(& & (d-c)/(a_1-b_1) \\ & + & ((b_2-a_2)/(a_1-b_1)) * x_2 \\ & + & \ldots \\ & + & ((b_n-a_n)/(a_1-b_1)) * x_n). \end{array}$$

In general, $solve(a = b)$ may contain variables that do not occur in $a = b$, and vice-versa.

There are a number of interesting canonizable and solvable theories including linear arithmetic, the theory of tuples and projections, algebraic datatypes like lists, set algebra, and the theory of fixed-sized bitvectors. In many cases, the canonizability and solvability of the union of theories (with disjoint signatures) follows from the canonizability and solvability of its constituent theories.[1] We do not address modularity issues here but instead assume that we already have a canonizer and solver for a single combined theory.

The solvers and canonizers characterized above are intended to work in the absence of uninterpreted function symbols. They are adapted to terms containing uninterpreted subterms by treating these subterms as variables. Canonizers are applied to terms containing uninterpreted subterms by renaming distinct uninterpreted subterms with distinct new variables. For a given term a, let γ be a bijective mapping between a set of variables X that do not appear in a and the uninterpreted subterms of a. The application of a substitution γ to a term a, written $\gamma[a]$, is defined so that $\gamma[a] = f(\gamma[a_1], \ldots, \gamma[a_n])$ if $a \equiv f(a_1, \ldots, a_n)$, where f is interpreted. If a is in the domain of γ, then $\gamma[a] = \gamma(a)$, and otherwise, $\gamma[a] = a$. Then $\sigma(a)$ is $\gamma[\sigma(\gamma^{-1}[a])]$.

For solving equalities containing uninterpreted terms, we introduce, as with σ, a bijective map γ between a set of variables X not occurring in a or b, and the uninterpreted subterms of a and b, such that

$$solve(a = b) = \gamma[solve(\gamma^{-1}[a] = \gamma^{-1}[b])] \ .$$

When uninterpreted terms are handled as above, the conditions in Definitions 3.1 and 3.4 must be suitably adapted by using $solvables(a)$ instead of $vars(a)$.

The proof theory for equality is augmented for canonizable, solvable theories by the proof rules:

1. Canonization: $\dfrac{}{T \vdash a = \sigma(a)}$, for any term a.
2. Solve: $\dfrac{T \vdash a = b \quad T \cup S \vdash c = d}{T \vdash c = d}$ if $S = solve(a = b) \neq \bot$ and $vars(c = d) \subseteq vars(T)$.
3. Solve-$\bot$: $\dfrac{T \vdash a = b}{T \vdash \mathit{false}}$, if $solve(a = b) = \bot$.

A sequent $T \vdash c = d$ is derivable if there is a proof of $T \vdash c = d$ using one of the inference rules: axiom, reflexivity, symmetry, transitivity, congruence, canonization, solve, or solve-$\bot$. We say that $T \vdash S$ is derivable if $T \vdash c = d$ is derivable for every $c = d$ in S. The sequent $T, S \vdash c = d$ is just $T \cup S \vdash c = d$. The *weakening* and *cut* lemmas below are easily verified.

[1] The general result on combining solvers claimed by Shostak [Sho84] is incorrect, but there are some restricted results on combining equational unifiers [BS96] that can be applied here.

Lemma 3.5 (weakening) *If $T \subseteq T'$ and $T \vdash a = b$ is derivable, then $T' \vdash a = b$ is derivable.*

Lemma 3.6 (cut) *If $T' \vdash T$ and $T \vdash a = b$ is derivable, then $T' \vdash a = b$ is derivable.*

Theorem 3.7 (proof soundness) *If $T \vdash a = b$ is derivable, then for any σ-model M and assignment ρ over $vars(T)$, $M, \rho \models T \vdash a = b$.*

Proof. By induction on the derivation of $T \vdash a = b$. The soundness of the *solve* rules follows from the conditions in Definition 3.4. ∎

A set of equalities S is said to be *functional* (in a left-to-right reading of the equality) if whenever $a = b \in S$ and $a = b' \in S$, $b \equiv b'$. For example, the solution set returned by *solve* is functional. A functional set of equalities can be treated as a substitution and the associated operations are defined below. $S(a)$ returns the solution for a if it exists in S, and a itself, otherwise. If $a = b$ is in S for some b, then a is in the domain of S, i.e., $dom(S)$.

$$\begin{aligned} S(a) &= \begin{cases} b & \text{if } a = b \in S \\ a & \text{otherwise} \end{cases} \\ dom(S) &= \{a \mid \exists b.\ a = b \in S\}. \end{aligned}$$

The operation $a \overset{S}{\sim} b$ checks if a is congruent to b in S, i.e., $a \equiv f(a_1, \ldots, a_n)$, $b \equiv f(b_1, \ldots, b_n)$, and $S(a_i) \equiv S(b_i)$ for $1 \leq i \leq n$. A set of equalities S is said to be *congruence-closed* when for any terms a and b in $dom(S)$ such that $a \overset{S}{\sim} b$, we have $S(a) \equiv S(b)$.

$S[a]$ replaces a subterm b in a by $S(b)$, where $b \in solvables(a)$.

$$\begin{aligned} S[f(a_1, \ldots, a_n)] &= f(S[a_1], \ldots, S[a_n]), \\ &\quad \text{if } f \text{ is interpreted} \\ S[a] &= S(a), \text{ otherwise.} \end{aligned}$$

$norm(S)(a)$ is a normal form for a with respect to S and is defined as $\sigma(S[a])$. The operation *norm* does not appear in Shostak's algorithm and is the key element of our algorithm and its proof. With S fixed, we use $\hat{a}$ as a syntactic abbreviation for $norm(S)(a)$.

$$norm(S)(a) = \sigma(S[a]).$$

Lemma 3.8 *If $solve(a = b) = S \neq \bot$, then $norm(S)(a) \equiv norm(S)(b)$.*

Proof. By definitions 3.3 and 3.4(2), for any σ-model M and assignment ρ', we have $M, \rho' \models S \iff M, \rho' \models a = b$. Let $a' \equiv S[a]$ and $b' \equiv S[b]$. By induction on a, $M, \rho' \models a = a'$, and similarly $M, \rho' \models b = b'$. Hence, $M, \rho' \models a' = b'$. Then, since M is a σ-model, by Definition 3.2, it must be the case that $\sigma(a') \equiv \sigma(b')$, and therefore $norm(S)(a) \equiv norm(S)(b)$. ∎

The definition of the *lookup* operation uses Hilbert's epsilon operator, indicated by the keyword *when*, to return $S(f(b_1, \ldots, b_n))$ when $b_1, \ldots, b_n$ satisfying the listed conditions can be found. If no such $b_1, \ldots, b_n$ can be found, then $lookup(S)(a)$ returns a itself. We show later that the *lookup* operation is used only when the results of this choice are deterministic.

$$\begin{aligned} lookup(S)(f(a_1, \ldots, a_n)) &= S(f(b_1, \ldots, b_n)), \\ &\quad \textit{when } b_1, \ldots, b_n : \\ &\quad f(b_1, \ldots, b_n) \in dom(S), \\ &\quad \text{and } a_i \equiv S(b_i), \\ &\quad \text{for } 1 \leq i \leq n \\ lookup(S)(a) &= a, \text{ otherwise.} \end{aligned}$$

$can(S)(a)$ is a canonical form in which any uninterpreted subterm e that is congruent to a known left-hand side e' in S is replaced by $S(e')$. It is analogous to the *canon* operation in Shostak's algorithm. We use $\overline{a}$ as a syntactic abbreviation for $can(S)(a)$.

$$\begin{aligned} can(S)(f(a_1, \ldots, a_n)) &= lookup(S)(f(\overline{a_1}, \ldots, \overline{a_n})), \\ &\quad \text{if } f \text{ is uninterpreted} \\ can(S)(f(a_1, \ldots, a_n)) &= \sigma(f(\overline{a_1}, \ldots, \overline{a_n})), \\ &\quad \text{if } f \text{ is interpreted} \\ can(S)(a) &= S(a), \text{ otherwise.} \end{aligned}$$

Lemma 3.9 (σ-norm) *If S is functional, then $norm(S)(\sigma(a)) \equiv \hat{a}$ and $can(S)(\sigma(a)) \equiv \overline{a}$.*

Proof. We know that $\vdash \sigma(a) = a$. Then for $b' = S[\sigma(a)]$ and $b = S[a]$, the equality $b' = b$ is valid in every σ-model. Then by Definition 3.2, $\sigma(S[\sigma(a)]) \equiv \sigma(S[a])$, and hence the first part of the theorem.

The reasoning in the second part is similar. If we let $R = \{b = \overline{b} \mid b \in [\![a]\!]\}$, then $can(S)(a) \equiv norm(R)(a)$. We can therefore use the first part of the theorem to establish the second part. ∎

We next introduce a composition operation for merging the results of a *solve* operation into an existing solution set. When $R \circ S$ is used, S must be functional, and the result contains $a = \hat{b}$ for each equality $a = b$ in R in addition to the equalities in S.

$$R \circ S = \{a = \hat{b} \mid a = b \in R\} \cup S.$$

The following lemmas about composition are given without proof.

Lemma 3.10 (norm decomposition) *If $R \cup S$ is functional, then*

$$norm(R \circ S)(a) \equiv norm(S)(norm(R)(a)).$$

$$
\begin{array}{rcl}
process(\{a = b, T\}) & = & assert(a = b, process(T)) \\
process(\emptyset) & = & \emptyset. \\
\\
assert(a = b, \bot) & = & \bot \\
assert(a = b, S) & = & cc(merge(\overline{a}, \overline{b}, S^+)), \text{ where,} \\
 & & S^+ = expand(S, \overline{a}, \overline{b}). \\
\\
expand(S, a, b) & = & S \cup \{e = e \mid e \in new(S, a, b)\}. \\
\\
new(S, a, b) & = & [\![a = b]\!] - dom(S). \\
\\
merge(a, b, S) & = & \bot, \text{if } solve(a = b) = \bot \\
merge(a, b, S) & = & S \circ solve(a = b), \text{ otherwise.} \\
\\
cc(\bot) & = & \bot \\
cc(S) & = & cc(merge(S(a), S(b), S)), \\
 & & when\ a, b: \\
 & & a, b \in dom(S) \\
 & & a \overset{S}{\sim} b, \text{ and } S(a) \not\equiv S(b) \\
cc(S) & = & S, \text{ otherwise.}
\end{array}
$$

Figure 1: Main Procedure: *process*

Lemma 3.11 (associativity of composition) *If $Q \cup R \cup S$ is functional, then*

$$(Q \circ R) \circ S = Q \circ (R \circ S).$$

Lemma 3.12 (monotonicity) *If $R \cup S$ is functional, then if $R(a) \equiv R(b)$, then $(R \circ S)(a) \equiv (R \circ S)(b)$, for any a and b.*

4 An Algorithm for Deciding Equality in the Presence of Theories

We next present an algorithm for deciding $T \vdash c = d$ for terms containing uninterpreted function symbols and function symbols interpreted in a canonizable and solvable theory. The algorithm for verifying $T \vdash c = d$ checks that $can(S)(c) \equiv can(S)(d)$, where $S = process(T)$. The *process* procedure shown in Figure 1, is written as a functional program. It is a mathematical description of the algorithm and not an optimized implementation. The *state* of the algorithm consists of a set of equalities S which holds the solution set. We demonstrate as an invariant that S is functional. Two terms a and b in $dom(S)$ are in the same equivalence class according to S if $S(a) \equiv S(b)$.

The operation $process(T)$ returns a final solution set by starting with an empty solution set and successively processing each equality $a = b$ in T by invoking $assert(a = b, S)$, where S is the state as returned by the recursive call of *process*. The invocation of $assert(a = b, S)$ is executed by first reducing a and b to their respective canonical forms $\overline{a}$ and $\overline{b}$. Next, S is expanded to include $e = e$ for each subterm e of $\overline{a} = \overline{b}$ where $c \notin dom(S)$. This preprocessing step ensures that S contains entries corresponding to any terms that might be needed in the congruence closure phase in the operation cc.[2] The *merge* operation then solves the equality $\overline{a} = \overline{b}$ to get a solution[3] S', and returns $S \circ S'$ as the new value for the state S. As we will show, this new value affirms $a = b$, but it is not congruence-closed and hence does not contain all the consequences of the assertion $a = b$. The step $cc(S)$ computes the congruence closure of S by repeatedly picking a pair of congruent terms a and b from $dom(S)$ such that $S(a) \not\equiv S(b)$ and merging them using $merge(S(a), S(b), S)$. Eventually either a contradiction is found or all congruent left-hand sides in S are merged and the cc operation terminates returning a congruence-closed solution set.

The above algorithm fixes the nontermination and incompleteness problems in Shostak's algorithm by introducing the *norm* operation and the composition operator $R \circ S$ to fold in a solution. The *norm* operation ensures that no new uninterpreted terms are introduced during congruence closure in the function cc, as is needed to guarantee termination. The composition operator $R \circ S$ ensures that any newly generated solution S is immediately substituted into R and the algorithm never attempts to find a solution for an already solved non-interpreted term.

We first illustrate the algorithm on some examples. The first example contains no interpreted symbols.

Example 4.1 Consider the goal $f^5(x) = x, f^3(x) = x \vdash f(x) = x$. The value of S after the base case is $\emptyset$. After the preprocessing of $f^3(x) = x$ in *assert*, S is $\{x = x, f(x) = f(x), f^2(x) = f^2(x), f^3(x) = f^3(x)\}$. After merging $f^3(x)$ and x, S is $\{x = x, f(x) = f(x), f^2(x) = f^2(x), f^3(x) = x\}$. When $f^5(x) = x$ is preprocessed in *assert*, $can(S)(f^5(x))$ yields $f^2(x)$ since $S(f^3(x)) \equiv x$, and S is left unchanged. When $f^2(x)$ and x have been merged, S is $\{x = x, f(x) = f(x), f^2(x) = x, f^3(x) = x\}$. Now $f(x) \overset{S}{\sim} f^3(x)$ and hence $f(x)$ and x are merged so that S is now $\{x = x, f(x) = x, f^2(x) = x, f^3(x) = x\}$.

[2] Actually, the interpreted subterms of $\overline{a} = \overline{b}$ need not all be included in $dom(S)$. Only those that are immediate subterms of uninterpreted subterms in $\overline{a} = \overline{b}$ are needed.

[3] Any variables occurring in $solve(a = b)$ and not in $vars(a = b)$ must be fresh, i.e., they must not occur in the original conjecture or be generated by any other invocation of *solve*.

The conclusion $f(x) = x$ easily follows since $can(S)(f(x)) \equiv x \equiv can(S)(x)$.

Example 4.2 Consider $y + 1 = x$, $f(y) + 1 = y - 1$, $f(x-1) - 1 = x + 1 \vdash \mathit{false}$ which is a permutation of our earlier example. Starting with $S \equiv \emptyset$ in the base case, the preprocessing of $f(x-1) - 1 = x + 1$ causes the equation to be placed into canonical form as $-1 + f(-1 + x) = 1 + x$ and S is set to

$$\{\ 1 = 1, -1 = -1, x = x, -1 + x = -1 + x, \\ f(-1 + x) = f(-1 + x), 1 + x = 1 + x\}.$$

Solving $-1+f(-1+x) = 1+x$ yields $f(-1+x) = 2+x$, and S is set to

$$\{\ 1 = 1, -1 = -1, x = x, -1 + x = -1 + x, \\ f(-1 + x) = 2 + x, 1 + x = 1 + x\}.$$

No unmerged congruences are detected. Next, $f(y) + 1 = y - 1$ is asserted. Its canonical form is $1 + f(y) = -1 + y$, and once this equality is asserted, the value of S is

$$\{\ 1 = 1, -1 = -1, x = x, -1 + x = -1 + x, \\ f(-1 + x) = 2 + x, 1 + x = 1 + x, y = y, \\ f(y) = -2 + y, -1 + y = -1 + y, \\ 1 + f(y) = -1 + y\}.$$

Next $y + 1 = x$ is processed. Its canonical form is $1 + y = x$ and the equality $1 + y = 1 + y$ is added to S. Solving $y + 1 = x$ yields $x = 1 + y$, and S is reset to

$$\{\ 1 = 1, -1 = -1, x = 1 + y, -1 + x = y, \\ f(-1 + x) = 3 + y, 1 + x = 2 + y, y = y, \\ f(y) = -2 + y, -1 + y = -1 + y, \\ 1 + f(y) = -1 + y, 1 + y = 1 + y\}.$$

The congruence close operation *cc* detects the congruence $f(1 - y) \overset{S}{\sim} f(x)$ and invokes *merge* on $3 + y$ and $-2 + y$. Solving this equality $3 + y = -2 + y$ yields $\perp$ returning the desired contradiction.

5 Analysis

We describe the proofs of termination, soundness, and completeness, and also present a complexity analysis.

Key Invariants. The *merge* operation is clearly the workhorse of the procedure since it is invoked from within both *assert* and *cc*. Let $U(X)$ represent the set $\{a \in X \mid a \text{ uninterpreted}\}$ of uninterpreted terms in the set X. Let A be $solvables(a)$, B be $solvables(b)$, and $S' = merge(a, b, S)$, then assuming $U(A \cup B) \subseteq dom(S)$ and for all $c \in A \cup B$, $S(c) \equiv c$, the following properties hold of S' if they hold of S:

1. Functionality.
2. Subterm closure: S is *subterm-closed* if for any $a \in dom(S)$, $[\![a]\!] \subseteq dom(S)$.
3. Range closure: S is *range-closed* if for any $a \in dom(S)$, $U(solvables(S(a))) \subseteq dom(S)$, and for any $c \in solvables(S(a))$, $S(c) \equiv c$.
4. Norm closure: S is *norm-closed* if $S(a) \equiv norm(S)(a)$ for a in $dom(S)$. This of course holds trivially for uninterpreted terms a.
5. Idempotence: S is *idempotent* if $S[S(a)] \equiv S(a)$, $norm(S)(S(a)) \equiv S(a)$, and $norm(S)(norm(S)(a)) \equiv norm(S)(a)$.

These properties can be easily established by inspection. Since whenever $merge(a, b, S)$ is invoked in the algorithm, the arguments do satisfy the conditions $U(A \cup B) \subseteq dom(S)$ and for all $c \in A \cup B$, $S(c) \equiv c$, it then follows that these properties are also preserved by *assert* and *cc*, and therefore hold of $process(T)$. We assume below that these invariants hold of S whenever the metavariable S is used with or without subscripts or superscripts.

Lemma 5.1 (merge equivalence) *Let $A = solvables(a)$ and $B \equiv solvables(b)$. Given that $U(A \cup B) \subseteq dom(S)$ and for all $c \in A \cup B$, $S(c) \equiv c$, if $S' = merge(a, b, S) \neq \perp$, then*

1. $norm(S')(a) \equiv norm(S')(b)$.
2. $U(dom(S')) = U(dom(S))$.

Proof. Let $R \equiv solve(a = b)$. By definition, $merge(a, b, S) \equiv S \circ R$. By Lemma 3.8, $norm(R)(a) \equiv norm(R)(b)$. Since $S(c) \equiv c$ for $c \in A \cup B$, $norm(S)(a) \equiv a$ and $norm(S)(b) \equiv b$. Hence, by *norm decomposition*, we have $norm(S')(a) \equiv norm(S')(b)$.

By Definition 3.4, $dom(R) \subseteq A \cup B$, hence $U(dom(S')) = U(dom(S))$. ∎

Termination. We define $\#(S)$ to represent the number of distinct equivalence classes partitioning $U(dom(S))$ as given by $P(S)$.

$$\begin{aligned} E(S)(a) &= \{b \in U(dom(S)) \mid S(b) \equiv S(a)\} \\ P(S) &= \{E(S)(a) \mid a \in U(dom(S))\} \\ \#(S) &= |P(S)| \end{aligned}$$

The definition of $cc(S)$ terminates because the measure $\#(S)$ decreases with each recursive call. If in the definition of *cc*, $merge(S(a), S(b), S) = \bot$, then clearly *cc* terminates. Otherwise, let $S' = merge(S(a), S(b), S) \neq \bot$, for a and b in $dom(S)$ such that $S(a) \not\equiv S(b)$ and $a \overset{S}{\sim} b$. In this case a and b must be uninterpreted terms since for interpreted terms a and b, if $a \overset{S}{\sim} b$, then $S(a) \equiv S(b)$ by *norm closure*. By *merge equivalence*, $norm(S')(S(a)) \equiv norm(S')(S(b))$ and $U(dom(S')) = U(dom(S))$. By *monotonicity*, for any c and d such that $S(c) \equiv S(d)$, we have $S'(c) \equiv S(d)$, and therefore $\#(S') \leq \#(S)$. However, by *norm closure*, $S'(a) \equiv S'(b)$ so that $\#(S') < \#(S)$.

Soundness. The following lemmas establish the soundness of the operations *norm* and *can* with respect to S. *Substitution soundness* and *can soundness* are proved by a straightforward induction on a, and *norm soundness* is a simple consequence of *substitution soundness*.

Lemma 5.2 (substitution soundness)
If $vars(a) \subseteq vars(T \cup S)$, *then* $T, S \vdash a = a'$ *is derivable, for* $a' \equiv S[a]$.

Lemma 5.3 (norm soundness)
If $vars(a) \subseteq vars(T \cup S)$, *then* $T, S \vdash a = \hat{a}$ *is derivable.*

Lemma 5.4 (can soundness)
If $vars(a) \subseteq vars(T \cup S)$, *then* $T, S \vdash a = \bar{a}$ *is derivable.*

Lemma 5.5 (merge soundness)
If $S' = merge(a, b, S) \neq \bot$, *then if* $T, S \vdash a = b$, *and* $T, S' \vdash c = d$ *with* $vars(c = d) \subseteq vars(T \cup S)$, *then* $T, S \vdash c = d$. *Otherwise,* $merge(a, b, S) = \bot$, *and* $T, S \vdash \bot$.

Proof. If $S' = merge(a, b, S) \neq \bot$, then let $R = solve(a = b)$. By *norm soundness*, $S, R \vdash S'$, and hence by *cut*, $T, S, R \vdash c = d$ is derivable. By the *solve* rule, $T, S \vdash c = d$ is derivable.

If $merge(a, b, S) = \bot$, then by similar reasoning using the *solve*-$\bot$ rule, $T, S \vdash false$ is derivable. ∎

Lemma 5.6 (cc soundness) *If* $S' = cc(S) \neq \bot$, $T, S' \vdash a = b$ *for* $vars(a = b) \subseteq vars(T, S)$, *then* $T, S \vdash a = b$ *is derivable. Otherwise,* $cc(S) = \bot$, *and* $S \vdash false$ *is derivable.*

Proof. By computation induction on the definition of *cc* using *merge soundness*. ∎

Lemma 5.7 (process soundness)
If $S = process(T_1) \neq \bot$, $T_1 \subseteq T_2$, *and* $T_2, S \vdash c = d$ *for* $vars(c = d) \subseteq vars(T_2)$, *then* $T_2 \vdash c = d$ *is derivable. Otherwise,* $process(T_1) = \bot$, *and* $T_1 \vdash false$ *is derivable.*

Proof. By induction on the length of T_1. In the base case, S is empty and the theorem follows trivially. In the induction step, with $T_1 = \{a = b, T_1'\}$ and $S' = process(T_1')$, we have the induction hypothesis that $T_2 \vdash c = d$ is derivable if $T_2, S' \vdash c = d$ is derivable, for any c, d such that $vars(c = d) \subseteq vars(T_2)$. We know by *can soundness* that $T_2, S' \vdash \overline{a} = a$ and $T_2, S' \vdash \overline{b} = b$ are derivable. When S' is augmented with identities over subterms of $\overline{a}$ and $\overline{b}$ to get S'^{+}, we have the derivability of $T_2, S' \vdash S'^{+}$. By *cc soundness*, we then have the derivability of $T_2, S'^{+} \vdash c = d$ from that of $T_2, S \vdash c = d$. The derivability of $T_2, S' \vdash c = d$ then follows by *cut* from that of $T_2, S'^{+} \vdash c = d$, and we get the conclusion $T_2 \vdash c = d$ by the induction hypothesis.

A similar induction argument shows that when $process(T_1) = \bot$, then $T_2 \vdash false$. ∎

Theorem 5.8 (soundness) *If* $S = process(T) \neq \bot$, $vars(a = b) \subseteq vars(T)$, *and* $\overline{a} \equiv \overline{b}$, *then* $T \vdash a = b$ *is derivable. Otherwise,* $process(T) = \bot$, *and* $T \vdash false$ *is derivable.*

Proof. If $S = process(T) \neq \bot$, then by *can soundness*, $T, S \vdash a = \overline{a}$ and $T, S \vdash b = \overline{b}$ are derivable. Hence, by transitivity and symmetry, $T, S \vdash a = b$ is derivable. Therefore, by *process soundness*, $T \vdash a = b$ is derivable.

If $process(T) = \bot$, then already by *process soundness*, $T \vdash false$. ∎

Completeness. We show that when $S = process(T)$ then $can(S)$ is a σ-model satisfying T. When this is the case, completeness follows from *proof soundness*. In proving completeness, we exploit the property that the output of *process* is congruence-closed.

Lemma 5.9 (confluence)
If S *is congruence-closed and* $U(\llbracket a \rrbracket) \subseteq dom(S)$, *then* $can(S)(a) \equiv norm(S)(a)$.

Proof. The proof is by induction on a. In the base case, when a is a variable, $can(S)(a) \equiv S(a) \equiv norm(S)(a)$.

If a is uninterpreted and of the form $f(a_1, \ldots, a_n)$, then $can(S)(a) \equiv lookup(S)(f(\overline{a_1}, \ldots, \overline{a_n}))$. Since S is *subterm-closed*, by the induction hypothesis and *norm closure*, we have $\overline{a_i} \equiv \hat{a_i} \equiv S(a_i)$ for $0 < i \leq n$. Then

there must be some b of the form $f(b_1, \ldots, b_n)$ such that $S(b_i) \equiv S(a_i)$, for $0 < i \leq n$, since a itself is such a b. Then by *congruence closure* and *norm closure*, $\overline{a} \equiv S(b) \equiv S(a) \equiv \hat{a}$, since $a \overset{S}{\sim} b$.

If a is interpreted, by the induction hypothesis and *subterm closure*, $\overline{a} \equiv \sigma(f(\overline{a_1}, \ldots, \overline{a_n})) \equiv \sigma(f(\hat{a_1}, \ldots, \hat{a_n})) \equiv \hat{a}$. ∎

Lemma 5.10 (can composition) *If $S' = S \circ R$ and S' is congruence-closed, then $can(S')(can(S)(a)) \equiv can(S')(a)$.*

Proof. By induction on a. When a is a variable. $can(S)(a) \equiv S(a)$. If $a \notin dom(S)$, then $S(a) = a$, and hence the conclusion. Otherwise, by range-closure, $U(\llbracket S(a) \rrbracket) \subseteq dom(S) \subseteq dom(S')$. Then, by *confluence*, *norm decomposition*, and *idempotence*, $can(S')(S(a)) \equiv norm(S')(S(a)) \equiv norm(R)(norm(S)(S(a))) \equiv norm(R)(norm(S)(a)) \equiv norm(S')(a) \equiv can(S')(a)$.

In the induction step, let $a \equiv f(a_1, \ldots, a_n)$. If a is uninterpreted, then if

$$f(\overline{a_1}, \ldots, \overline{a_n}) \overset{S}{\sim} f(b_1, \ldots, b_n)$$

for some $f(b_1, \ldots, b_n) \in dom(S)$, then $\overline{a} \equiv S(f(b_1, \ldots, b_n))$. The reasoning used in the base case can then be repeated to derive the conclusion. Otherwise, $\overline{a} \equiv f(\overline{a_1}, \ldots, \overline{a_n})$ and by the induction hypothesis and the definition of *can*, $can(S')(\overline{a}) \equiv lookup(S')(f(can(S')(a_1), \ldots, can(S')(a_n))) \equiv can(S')(a)$.

When a is interpreted, by the induction hypothesis and the *σ-norm* lemma,

$$\begin{array}{ll} & can(S')(\overline{a}) \\ \equiv & can(S')(\sigma(f(\overline{a_1}, \ldots, \overline{a_n}))) \\ \equiv & \sigma(f(can(S')(\overline{a_1}), \ldots, can(S')(\overline{a_n}))) \\ \equiv & can(S')(a). \end{array}$$

∎

Lemma *can composition* with $\emptyset$ for R yields the idempotence of $can(S)$ for congruence-closed S so that we can define a σ-model M_S in terms of $can(S)$. The domain D of M_S consists of $\{a | can(S)(a) = a\}$. The mapping of functions is such that $M_S(f)(\mathbf{a_1}, \ldots, \mathbf{a_n}) = lookup(S)(f(\mathbf{a_1}, \ldots, \mathbf{a_n}))$, if f is uninterpreted. If f is interpreted $M_S(f)(\mathbf{a_1}, \ldots, \mathbf{a_n}) = \sigma(f(\mathbf{a_1}, \ldots, \mathbf{a_n}))$. If $\rho[x] = \rho(x)$ and $\rho[f(a_1, \ldots, a_n)] = f(\rho[a_1], \ldots, \rho[a_n])$, then by the idempotence of $can(S)$, $M_S\llbracket a \rrbracket_\rho$ is just $can(S)(\rho[a])$. Lemma σ-norm can then be used to show $M_S \models \sigma(a) = a$. M_S is therefore a σ-model. Correspondingly, for a given set of variables X, ρ_S^X is defined so that $\rho_S^X(x) = can(S)(x)$ for $x \in X$.

Lemma 5.11 (can σ-model) *If $S = process(T) \neq \bot$ and $X = vars(T)$, then $M_S, \rho_S^X \models a = b$ for any $a = b \in T$.*

Proof. Showing that $M_S, \rho_S^X \models a = b$ is the same as showing that $can(S)(a) \equiv can(S)(b)$. The proof is by induction on T. In the base case, T is empty. In the induction step, $T = \{a = b, T'\}$ with $X' = vars(T')$. Let $S' = process(T')$. By the induction hypothesis, $M_{S'}, \rho_{S'}^{X'} \models T'$. With $S'^{+} = expand(S, a', b')$ for $a' \equiv can(S')(a)$ and $b' \equiv can(S')(b)$, let $S_0 = merge(a, b, S'^{+})$, hence by *merge equivalence*, $norm(S_0)(a') \equiv norm(S_0)(b')$. By associativity of composition, it can be shown that there is an R such that $S = S_0 \circ R$ and an R' such that $S = S'^{+} \circ R'$. Hence by monotonicity, $norm(S)(a') \equiv norm(S)(b')$. Since S is congruence-closed, by *confluence*, $can(S)(a') \equiv norm(S)(a')$ and $can(S)(b') \equiv norm(S)(b')$. Hence, $can(S)(a') \equiv can(S)(b')$.

It can also be shown that $can(S'^{+})(a) \equiv can(S')(a)$, and similarly for b. Therefore, by *can composition*, we have $can(S)(a) \equiv can(S)(b)$, and hence $M_S, \rho_S^X \models a = b$. A similar argument shows that for $c = d \in T'$, since $can(S')(c) \equiv can(S')(d)$, we also have $can(S)(c) \equiv can(S)(d)$. ∎

When $T \vdash false$ is derivable, we know by *proof soundness* that there is no σ-model satisfying T and hence by the *can σ-model* lemma, $process(T)$ must be $\bot$.

Theorem 5.12 (completeness) *If $S = process(T) \neq \bot$ and $T \vdash a = b$, then $can(S)(a) \equiv can(S)(b)$.*

Proof. Since $M_S, \rho_S^X \models T$ by *can σ-model* for $X = vars(T)$, we have by *proof soundness* that $can(S)(a) \equiv can(S)(b)$. ∎

Complexity. We have already seen in the termination argument that the number of iterations of *cc* in *process* is bounded by the number of distinct equivalence classes of terms in $dom(S)$ which is no more than the number of distinct uninterpreted terms. We will assume that the *solve* operation is performed by an oracle and that there is some fixed bound on the size of the solution set returned by it. In the case of linear arithmetic, the solution set has at most one element. Let n represent the number of distinct terms appearing in T which is also a bound on $|S|$ and on the size of the largest term appearing in S. The composition operation can be implemented in linear time. Thus the entire algorithm has $O(n^2)$ steps assuming that the σ and *solve* operations are length-preserving and ignoring the time spent inside *solve*.

6 Conclusions

Shostak's decision procedure for equality in the presence of interpreted and uninterpreted functions is seriously flawed. It is both incomplete and nonterminating, and hence not a decision procedure. All subsequent variants of Shostak's algorithm have been similarly flawed. This is unfortunate because decision procedures based on Shostak's algorithm are at the core of a number of widely used verification systems. We have presented a correct algorithm that captures Shostak's key insights, and described proofs of termination, soundness, and completeness.

Acknowledgments: We are especially grateful to Clark Barrett for instigating this work and correcting several significant errors in earlier drafts, and to Jean-Christophe Filliâtre for his oCaml implementation which yielded useful feedback on the algorithm studied here. The presentation has been substantially improved thanks to the suggestions of the anonymous referees and those of Nikolaj Bjørner, David Cyrluk, Bruno Dutertre, Ravi Hosabettu, Pat Lincoln, Ursula Martin, David McAllester, Sam Owre, John Rushby, and Ashish Tiwari.

References

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201, Palo Alto, CA, November 1996. Springer-Verlag.

[Bjø99] Nikolaj Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Stanford University, 1999.

[BRRT99] L. Bachmair, C. R. Ramakrishnan, I.V. Ramakrishnan, and A. Tiwari. Normalization via rewrite closures. In *International Conference on Rewriting Techniques and Applications, RTA '99*, Berlin, 1999. Springer-Verlag.

[BS96] F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *J. Symbolic Computation*, 21:211–243, 1996.

[CLS96] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[DST80] P.J. Downey, R. Sethi, and R.E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, 1980.

[EHD93] Computer Science Laboratory, SRI International, Menlo Park, CA. *User Guide for the* EHDM *Specification Language and Verification System, Version 6.1*, February 1993. Three volumes.

[FORS01] J-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonizer and solver. In *CAV 01: Computer-Aided Verification*. Springer-Verlag, 2001. To appear.

[Kap97] Deepak Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *International Conference on Rewriting Techniques and Applications, RTA '97*, number 1232 in Lecture Notes in Computer Science, pages 23–37, Berlin, 1997. Springer-Verlag.

[MT96] Zohar Manna and The STeP Group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[SSMS82] R. E. Shostak, R. Schwartz, and P. M. Melliar-Smith. STP: A mechanized logic for specification and verification. In D. Loveland, editor, *6th International Conference on Automated Deduction (CADE)*, volume 138 of *Lecture Notes in Computer Science*, New York, NY, 1982. Springer-Verlag.

Appears in the Proceedings of CADE'02, LNCS. ©Springer-Verlag

Formal Verification of a Combination Decision Procedure*

Jonathan Ford and Natarajan Shankar

Computer Science Laboratory
SRI International, Menlo Park CA 94025 USA
{ford, shankar}@csl.sri.com
Phone: (650)859-5272

Abstract. Decision procedures for combinations of theories are at the core of many modern theorem provers such as ACL2, EHDM, PVS, SIMPLIFY, the Stanford Pascal Verifier, STeP, SVC, and Z/Eves. Shostak, in 1984, published a decision procedure for the combination of canonizable and solvable theories. Recently, Ruess and Shankar showed Shostak's method to be incomplete and nonterminating, and presented a correct version of Shostak's algorithm along with informal proofs of termination, soundness, and completeness. We describe a formalization and mechanical verification of these proofs using the PVS verification system. The formalization itself posed significant challenges and the verification revealed some gaps in the informal argument.

1 Introduction

Decision procedures play an important rôle in a number of areas such as automated deduction, computer-aided verification, and constraint solving. Since bugs in decision procedures can lead to unsound inferences, it is natural to ask if such verification tools can themselves be verified. We present here the first instance of a verified decision procedure for a combination of theories based on Shostak's ideas. Shostak's algorithm [Sho84] for building decision procedures for the union of canonizable and solvable equational theories has been widely used despite the lack of a convincing correctness proof. Recently, Ruess and Shankar [RS01] showed that this algorithm (even with minor flaws corrected [CLS96]) was both nonterminating and incomplete. They gave a corrected version of the algorithm along with informal proofs for termination, soundness, and completeness. We undertook the challenge of formalizing and verifying these informal arguments using the PVS verification system [ORS92]. The results of our verification are presented here along with observations regarding the challenges that we encountered in the formalization and verification process.

* This work was funded by NSF Grant CCR-0082560, DARPA/AFRL Contract F33615-00-C-3043, and NASA Contract NAS1-00079. Sam Owre, Harald Rueß, and John Rushby of SRI provided insightful comments on earlier drafts. We thank the anonymous referees for their constructive criticism.

The correctness of decision procedures has been an important theme in automated reasoning. Several approaches have been developed for using decision procedures to gain efficiency in proof construction without compromising soundness. The LCF approach [GMW79] admits only those decision procedures that can be introduced as *tactics*, which are metalanguage operations for reducing proof goals to subgoals in a way that is justifiable in terms of the primitive inferences of the object logic. Tactics can be hard to define (since they have to mimic proof steps) and inefficient (since they have to generate low-level inference steps). The generation of *proof objects* from finished proofs is another way of ensuring that each proof can be constructed using only the primitive inference steps. The construction of proof objects even from finished proofs can be inefficient in both time and space.

In order to avoid the inefficiency of fully expansive proof generation, a number of researchers have advocated the verification of decision procedures. Boyer and Moore [BM81] introduce a notion of metafunctions, i.e., function definitions in the object logic that could be applied to object logic expressions. They use computational reflection to capture the meanings of these expressions in the object logic and verify the soundness of some simple derived inference rules in this manner. Boyer and Moore [BM79] also verified the semantic correctness of a tautology checker for conditional expressions. Shankar [Sha85] verified both the semantic and proof-theoretic correctness of a tautology checker for propositional logic. Some recent examples of verified decision procedures include a Coq verification of a Gröbner basis algorithm for membership in polynomial ideals by Théry [Thé98], the verification of ordered binary decision diagram (OBDD) operations using PVS by von Henke, Pfab, Pfeifer, and Ruess [vHPPR98], and a similar Coq verification of OBDD operations by Verma and Goubault [VGL00]. Both the algorithm and the theory underlying the combination decision procedure considered here are significantly more complex than these previously verified decision procedures.

The primary contribution of our work is in demonstrating the feasibility of formally verifying complex decision procedures. The variant of Shostak's algorithm we have verified is quite recent and its foundations are not widely understood. Our verification closely follows the published informal proof [RS01] so that we could directly assess its validity. We also used details from an unpublished report that included proofs of some of the lemmas that were given without proof in the published paper. The verification exposed some gaps in the informal argument. We found a monotonicity claim in the informal argument to be false without qualification, but only the qualified form was actually used. A step that is hinted at as being routine, turned out to not be all that obvious. In the algorithm, any solution returned by the solver must contain variables that are either from the given equality or are "fresh". Making the notion of freshness precise, and working with this constraint proved to be one of the major challenges in the formal verification. The verification makes very heavy use of the PVS type system. Our use of PVS types exposed some of the weaknesses in a type propagation feature of the language called *typing judgements*.

Since PVS itself employs Shostak's method (with the incompleteness and nontermination bugs), the validity of this verification might be called into question. However, the Shostak procedure used in PVS is not known to be unsound. Future versions of PVS will employ the ICS decision procedures [FORS01] that are based on the theory verified here. Despite the circularity between the verifier and the verified program, this kind of verification is still quite useful. An unsuccessful proof attempt might reveal significant bugs. A successful verification of the decision procedures could be certified through proof-object generation but subsequently used without the supporting proof objects.

The decision procedure as verified here is not executable, but it is possible to derive a verified, executable version that can be turned into efficient Common Lisp code [Sha99]. The code generated from the verified decision procedure is unlikely to be as efficient as the highly optimized ICS implementation, but it could still be used as a reference procedure that can be invoked when certified results are needed.

We verify both soundness and completeness. The completeness property is crucial. Higher-level simplification routines might diverge or behave erroneously if they incorrectly assume completeness. Due to its complexity and popularity, the verification of Shostak's algorithm is a good case-study for assessing the feasibility of certifying decision procedures.

2 Shostak's Algorithm

We focus here on the verification of a decision procedure for equational theories where terms are constructed from a combination of interpreted and uninterpreted function symbols. There are two basic methods for building decision procedures for combinations of disjoint theories. Nelson and Oppen's method [NO79] combines decision procedures for the individual theories by allowing them to share specific kinds of equality information. Shostak's method [Sho84] extends congruence closure to equational theories that are canonizable and solvable. Nelson and Oppen's method is more generally applicable, but Shostak's method has certain advantages. It is an online algorithm, i.e., processes inputs incrementally, so that the term universe for the input is not known in advance. It also yields a useful function for computing a canonical form respecting the given input equalities.

All formulas are equalities between terms which are constructed from variables by means of n-ary function application for $n \geq 0$. Sequents of the form $T \vdash a = b$ assert the implication between the antecedent equalities in the set T and the consequent equality $a = b$. The basic theory of equality with all function symbols uninterpreted, i.e., without any fixed interpretation, is decidable by means of *congruence closure*. Shostak's algorithm extends the congruence closure decision procedure to handle interpreted operations from a *canonizable* and *solvable* theory. Informally, a theory is canonizable if there is a canonizer operation σ such that $\sigma(a) \equiv \sigma(b)$ exactly when $a = b$ is valid in the theory. It

is solvable if there is an operation *solve* such that $solve(a = b)$ either returns $\perp$ when $a = b$ is unsatisfiable, or a solved form S that is equivalent to $a = b$.

Shostak's procedure takes as parameters, a solver *solve* and canonizer σ for a theory such as linear arithmetic. The algorithm verifies a sequent $T \vdash a = b$ by processing the equalities in T to build a solution set S of equalities in solved form, or to return $\perp$ indicating that a contradiction was found in T. If a solution set S is returned, then one can use S and σ to define a canonizer *can* such that $can(S)(f(e))$ returns $\sigma(f(can(S)(e)))$ if f is interpreted. If f is uninterpreted, $can(S)(f(e))$ returns c' for some c equivalent to $f(can(S)(e))$ where $c = c'$ is in S. The conclusion equality $a = b$ can be tested for validity by checking if $can(S)(a) \equiv can(S)(b)$. The operation $can(S)$ is also used for preprocessing the input equalities from T. The preprocessed input equalities are solved and the solution (if any) is composed with the existing value of S. The solution set S is maintained in congruence-closed form so that the right-hand sides of congruent left-hand side terms are merged by solving the equality between them and merging the results into S.

The theory of linear arithmetic is a typical example of a canonizable and solvable theory. A canonizer can be given by means of a function that returns an ordered sum-of-products representation for a given linear polynomial by merging monomials over the same variable into a single monomial. A solver can be given by using algebraic manipulations to isolate a variable on the left-hand side. The Shostak procedure of Ruess and Shankar [RS01] can be illustrated on the sequent

$$f(x-1)-1 = f(y)+1,\ y-x+1 = 0 \vdash \mathit{false},$$

where $+$, $-$, and the numerals are from the theory of linear arithmetic, *false* is an abbreviation for $0 = 1$, and f is an uninterpreted function symbol. Starting with $S \equiv \emptyset$ in the base case, the preprocessing of $f(x-1)-1 = f(y)+1$ causes the equality to be placed into canonical form as $-1 + f(-1 + x) = 1 + f(y)$. The solution set S is initialized to contain reflexivity statements for the non-interpreted subterms in the canonicalized input equality as $\{x = x, y = y, f(-1+x) = f(-1+x), f(y) = f(y)\}$. Solving $-1 + f(-1 + x) = 1 + f(y)$ yields $f(-1+x) = 2+f(y)$, and S is set to $\{x = x, y = y, f(-1+x) = 2+f(y), f(y) = f(y)\}$. No unmerged congruences are detected in S. Next, $y - x + 1 = 0$ is canonized as $1 - x + y = 0$, and solved as $x = 1 + y$. This solution is composed with S to yield $\{x = y + 1, y = y, f(-1 + x) = 2 + f(y), f(y) = f(y)\}$. The congruence between $f(-1 + x)$ and $f(y)$ is detected since the canonical form of $-1 + x$ is y when the solution for x is inserted and the result is canonized by σ. The procedure then tries to merge the respective solutions of $f(-1+x)$ and $f(y)$ by solving $2 + f(y) = f(y)$. The solver returns $\perp$ so that the original sequent is asserted to be valid.

As a second example, one can check that the sequent $f(x-1)-1 = f(y)+1 \vdash g(f(x-1)-2) = g(f(y))$ is valid by computing S to be $\{x = x, y = y, f(-1+x) = 2+f(y), f(y) = f(y)\}$, and verifying $can(S)(g(f(x-1)-2)) \equiv can(S)(g(f(y)))$.

3 Formalizing Shostak's Algorithm in PVS

A brief introduction to PVS is given in Appendix A. The formalization exploits several advanced features of the PVS language including recursive datatypes, predicate subtypes, dependent types, Hilbert's choice operator, and inductive relations. We describe the formalization in sufficient detail so that it can be checked for conformity with the informal arguments [RS01] (abbreviated below as **RS**) and reproduced using some other automated proof checker.[1]

Syntax. Terms are built from a given signature consisting of a set of variables X and function symbols F. A *term* is either a variable x for $x \in X$ or of the form $f(a_1, \ldots, a_n)$, where $f \in F$. A term of the form $f(a_1, \ldots, a_n)$ is *interpreted* (respectively, *uninterpreted*) if f is interpreted (respectively, uninterpreted). Terms are formalized by means of a recursive datatype `syntax` consisting of a constructor `v` for variables with a natural number index field `index`, and an application constructor `app` with a function symbol field `func` and an arguments field `args` which is formalized as a *dependent* type `[below(arity(func)) -> syntax]` which represents an *array* of `syntax` in the arity of the function symbol of the term. The type `below(num)` for a natural number `num` is the (possibly empty) subrange $0, \ldots, \texttt{num} - 1$.[2] The function symbol type `funsymbs` is also a datatype consisting of constructors `ifn` and `ufn` for interpreted and uninterpreted function symbols, respectively, each with an `index` field and an `arity` field, and a `thry` (theory) field for interpreted function symbols.

1
```
funsymbs: DATATYPE
  BEGIN
   IMPORTING theories
   ifn(index: nat, arity: nat, thry: TH): ifn?
   ufn(index: nat, arity: nat): ufn?
  END funsymbs

syntax: DATATYPE
  BEGIN
   IMPORTING funsymbs, max_lemmas
   v(index: nat): v?
   app(func: funsymbs,
       args: [below(arity(func)) -> syntax]): app?
  END syntax
```

Since we are admitting just one interpreted theory, we fix a theory `th`. The predicate `thry_func` checks that its argument is an interpreted function symbol

[1] The complete PVS 2.4.1 dump file is available at `ftp://ftp.csl.sri.com/pub/users/shankar/shostak-verification-dump`.

[2] An application could also be formalized in terms of a *list* of arguments whose length is the arity of the function symbol. The array-based formalization has some important advantages. Terms are well-formed by construction thus avoiding the need for cumbersome proof obligations. Operations on terms can be defined by a simple structural recursion without the use of mutual recursion on terms and lists of terms.

from theory `th`. The type `thry_func` is the predicate subtype corresponding to the predicate `thry_func`.

```
  thry_func(ff:funsymbs): bool =                                    2
    ifn?(ff) AND thry(ff) = th
```

The type of equalities is defined as a record type with fields `lhs` and `rhs`.

```
    equality: TYPE = [# lhs, rhs: syntax #]                         3
```

The variables `a`, `b`, and `c` are declared to range over terms, `aa`, `bb`, and `cc` range over equalities, and `R`, `S`, and `T` range over lists of equalities.

The set of variables in a term `a` is defined using datatype recursion as `vars(a)`. Sets are just predicates in the higher-order logic so that a variable `x` is in the set `vars(a)` iff `vars(a)(x)` holds. The set `vars(a)` can be shown to be finite by structural induction. A term a is well-typed in n for a natural number n, if the index of any variable in a is below n. This is represented by the predicate `well_typed?(n)(a)` and the corresponding type `typed(n)`. The operation of collecting the set of subterms of a given term is represented by `subterm(a)`. The definitions of these operations are omitted.

Pure Terms. The canonizer and solver are defined for *pure* terms, i.e., terms without uninterpreted function symbols, but then applied to arbitrary terms by treating the uninterpreted subterms as variables. We formalize pure terms by means of a datatype `pure` that has two classes of variables: `v(i)` for the ordinary variables indexed by `i`, and `u(a)` corresponding to the uninterpreted term `a`. Function applications for pure terms are typed to contain only interpreted function symbols. It is easy to define an operation `abs` that converts a term to the corresponding pure term, and its inverse `gamma`.

```
pure[(IMPORTING theories) th: TH]: DATATYPE WITH SUBTYPES var?, func?  4
BEGIN
   IMPORTING syntax_ops[th]
   v(index: nat): v? : var?
   u(a: uninterpreted): u? :var?
   app(func: thry_func,
       args: [below(arity(func)) -> pure]): app? : func?
END pure
```

Semantics. The semantics for a term a is given by $M[\![a]\!]\rho$ for an *interpretation* M over a domain D such that $M(f)$ yields a mapping from D^n to D for function symbol f of arity n, and an *assignment* ρ mapping variables to values in D. For variables, $M[\![x]\!]\rho = \rho(x)$, and $M[\![f(a_1, \ldots, a_n)]\!]\rho = M(f)(M[\![a_1]\!]\rho, \ldots, M[\![a_n]\!]\rho)$. We say that $M, \rho \models a = b$ iff $M[\![a]\!]\rho = M[\![b]\!]\rho$, and $M \models a = b$ iff $M, \rho \models a = b$ for all assignments ρ over $vars(a = b)$. An equality is *valid* if for all D, M: $M \models a = b$.

The concept of a valid equality requires quantification over all domains D and interpretations M over D. In PVS, such a domain would have to be introduced

as the type parameter of a theory. Since PVS does not admit quantification over types, the domain must be given as a subset or a subtype of a fixed type. We take this fixed type to be the set of all terms.[3] This type can be informally shown to be adequate for representating any domain set D for the purposes of equality. The assignment ρ is formalized as a mapping from the set of all variables to the domain D.

In the semantics for pure terms, the domain type `D` is the type of pure terms and a model is a dependent record type consisting of a domain field `mdom` that is a subset of `D`, and a function interpretation field `f` that is a dependent type mapping a function `ff` and an array of argument valuations to a valuation for the application. The type `arity(ff)` is an abbreviation for `below(arity(ff))`.

```
D: TYPE+ = pure                                                    5

model: TYPE = [# mdom : setof[D],
                 f: [ff: thry_func ->
                       [[arity(ff) -> (mdom)] -> (mdom)]] #]
```

Solutions. The "state" of the algorithm is maintained in a solution set S that is just a list of equalities of a special form. The operation `apply(S)(a)` (informally, $S(a)$) is defined recursively to look up the solution for `a` (if any) in `S`.[4]

```
apply(S)(a): RECURSIVE syntax =                                    6
 CASES S OF
  null: a,
  cons(aa, R): IF lhs(aa) = a
                THEN rhs(aa)
                ELSE apply(R)(a)
               ENDIF
 ENDCASES
MEASURE length(S)
```

The operation `replace_vars(S)(d)` (informally, $S[d]$) returns the result of replacing all occurrences of any left-hand side variable from `S` in a pure term `d`, by the corresponding right-hand side. The `replace_vars` operation is extended from pure terms to arbitrary terms as `replace_solvables`. The operation `subst(rho)(d)` (used in 7) is similar to `replace_vars(S)(d)` but `rho` here is a substitution mapping variables to terms.

Canonizers. A canonizer σ for pure terms from a theory τ is a parameter to the combination decision procedure. A valid canonizer is required to verify validities, i.e., $\models_\tau a = b$ implies $\sigma(a) \equiv \sigma(b)$, and additionally preserve variables, $\sigma(x) = x$ and $vars(\sigma(a)) \subseteq vars(a)$, be idempotent, $\sigma(\sigma(a)) = \sigma(a)$, and leave

[3] The type of closed terms, when nonempty, is also a valid candidate for the domain.

[4] The termination of the recursive definition is justified by the measure `length(S)` which causes the typechecker to generate proof obligations verifying that the measure decreases with each recursive call.

subterms canonical, $\sigma(b) = b$ for any subterm b of $\sigma(a)$. These conditions on a valid canonizer are captured by the predicate `canonizer?(sigma)`. The validity condition is awkward since it uses an oracle $\models_\tau$ for τ-validity. We found a way to replace this condition by the sufficient pair of conditions on σ:

1. σ-substitutivity: $\sigma(\rho[a]) \equiv \sigma(\rho[\sigma(a)])$, for any substitution ρ, and
2. σ-distributivity: $\sigma(f(\sigma(a_1), \ldots, \sigma(a_n))) \equiv \sigma(f(a_1, \ldots, a_n))$.

`canonical?(sigma)(a)` is defined to hold when `sigma(a) = a`.

7
```
canonizer?(sigma): bool =
  (   (FORALL d, rho: sigma(subst(rho)(d)) = sigma(subst(rho)(sigma(d))))
  AND (FORALL d: app?(d) IMPLIES
         sigma(app(func(d), LAMBDA (i:arity(func(d))): sigma(args(d)(i))))
         = sigma(d))
  AND (FORALL u   : sigma(u) = u)
  AND (FORALL d, u: vars(sigma(d))(u) IMPLIES vars(d)(u))
  AND (FORALL d   : sigma(sigma(d)) = sigma(d))
  AND (FORALL d, f: sigma(d) = f IMPLIES
         (FORALL (i:arity(func(f))): canonical?(sigma)(args(f)(i)))))
```

The adaptation of the canonizer from pure terms to terms is done through `gamma` and `abs`. The canonizer for arbitrary terms, `sig(a)` (used in 9 and 10), is defined as `gamma(sigma(abs(a)))`, where `sigma` is the given canonizer for pure terms. Model M is a *σ-model* if $M \models \sigma(a) = a$ for all a, and $a = b$ is *σ-unsatisfiable* (formalized as the PVS predicate `unsatisfiable`) if $M, \rho \not\models a = b$ for all M and ρ.

Solver. A solver *solve* is another parameter to the algorithm. A valid solver must be such that $solve(a = b)$ either returns $\perp$ when $a = b$ is σ-unsatisfiable, or returns a (possibly empty) list S of n equalities of the form $x_i = t_i$ for $1 \leq i \leq n$, where $x_i \in vars(a = b)$ $x_i \not\equiv x_j$ for $i \neq j$, $x_i \notin vars(t_j)$, t_i is canonical ($\sigma(t_i) = t_i$), for $1 \leq x, y \leq n$, and $a = b$ and S are σ-equivalent: for all σ-models M and assignments ρ over the variables in a and b, $M, \rho \models a = b$ iff there is an assignment ρ' extending ρ, over the variables in S, a, and b, such that $M, \rho' \models S$.

The notion of a *solution* for pure term equalities is formalized as the predicate `solve(n, dd, S)` for an index `n`, an equality `dd`, and a solution list `S`. The predicate checks that `dd` is satisfiable, the solution list of equalities `S` is a well-formed solution that is σ-equivalent (formalized as the PVS predicate `sig_equivalent?`) to `dd`. Any variables in `S` not in `dd` must be of index above `n`.

8
```
solve(n, dd, S): bool =
  IF unsatisfiable(dd) THEN
    FALSE
  ELSE
    new_vars_above(n, dd)(S) AND
    check_solution(dd)(S) AND
    sig_equivalent?(dd, S)
  ENDIF
```

A pure term solver is easily extended to one that works on terms. A given solver `solv` is typed so that `solv(m, dd)` returns a dependent record `r` with fields `n` and `s`, where `r`n` is an index that is at least `m` and `r`s` is either `bottom` or of the form `up(S)` for a solution list of equalities `S` that is well-typed in `r`n`.

Canonical Forms. The operation $norm(S)(a)$ (represented as `norm(S)(a)`) for a canonizer `sig`, is informally defined as $\sigma(S[a])$. The definition of `norm` is used to show that if `solve(m, aa, S)` holds, then `norm(S)(lhs(aa)) = norm(S)(rhs(aa))`, and to define the composition of two equality lists `R` and `S` as `R o S`.

```
norm(S)(a): syntax = sig(replace_solvables(S)(a))                    9

o(R, S): RECURSIVE eqlist =
 CASES R OF
  null: S,
  cons(aa, T): cons(eq(lhs(aa), norm(S)(rhs(aa))), T o S)
 ENDCASES
MEASURE length(R)
```

Since composition is defined recursively, its definition includes a termination measure `length(R)` that is used to generate termination proof obligations. The definitions above are used to prove the associativity of composition and the claim: `norm(R o S)(a) = norm(S)(norm(R)(a))`.

The operation `lookup(S)(a)` is defined so that if `a` is a variable, then it returns `apply(S)(a)` which is the formalization of $S(a)$. When a is an application, then `lookup` is defined to scan S till it finds an equality whose left-hand side is of the form $f(a_1, \ldots, a_n)$, where $f(norm(S)(a_1), \ldots, norm(S)(a_n)) \equiv a$.[5]

The canonizer `can(S)(a)` is then defined in terms of the `lookup` operation.

```
can(S)(a): RECURSIVE syntax =                                        10
 CASES a OF
  v(i): apply(S)(a),
  app(ff, args):
   IF intheory?(a) THEN
    sig(app(ff, LAMBDA (i:arity(ff)): can(S)(args(i))))
   ELSE
    lookup(S)(app(ff, LAMBDA (i:arity(ff)): can(S)(args(i))))
   ENDIF
 ENDCASES
MEASURE rank(a)
```

Congruence. Congruence with respect to a solution set S, $f(a_1, \ldots, a_n) \overset{S}{\sim} f(b_1, \ldots, b_n)$, is defined to hold exactly when $norm(S)(a_i) \equiv norm(S)(b_i)$ for $1 \leq i \leq n$. This is captured formally by the predicate `congruent(S)(a, b)`.

[5] This definition of `lookup` is slightly different from that of **RS** which uses $S(a_i)$ instead of $norm(S)(a_i)$. The **RS** definition requires keeping $dom(S)$ subterm-closed, whereas we only require closure under the uninterpreted subterms. Our definition is executable in contrast to the **RS** definition which uses Hilbert's epsilon operator.

```
congruent(S)(a, b): bool =                                          11
  app?(a) AND
  app?(b) AND
  func(a) = func(b) AND
  (FORALL (i:arity(func(a))):
    norm(S)(args(a)(i)) = norm(S)(args(b)(i)))
```

A solution set is *congruence-closed* when the right-hand sides corresponding to any pair of congruent left-hand sides are identical.

```
congruence_closed(S): bool =                                        12
  (FORALL (a,b:(dom(S))): congruent(S)(a, b) IMPLIES
                          apply(S)(a) = apply(S)(b))
```

The solution set that forms the "state" of the algorithm is typed to satisfy the invariants given by the predicate `invariants(S)`. These invariants assert that the left-hand sides of equalities in the solution set S must be variables or uninterpreted terms, the uninterpreted subterms of any equality S must in the domain of S, and any right-hand side term must be canonical, and $S(a)$ and $norm(S)(a)$ must coincide for any $a \in dom(S)$, among other conditions. The predicate `invariant(S)` is used to define a type `above_tinvariants(n)` which ensures that the state is a record `r` consisting of an index `r`n` and a solution set `r`s` which is either `bottom` or `up(S)`, where `S` is well-typed in `r`n` and satisfies `invariants(S)`.

The Main Procedure. The congruence closure operation `cc(r)` successively merges the right-hand sides corresponding to *chosen* congruent pairs of left-hand side terms in the solution set `r`s`. The operation `merge(m, aa, S)` (used in 13 and 14) computes `solv(m, aa)` as a record `r`, returning `bottom` if `r`s` is `bottom`, and the record `(# n := r`n, s := S o down(r`s)#)`, otherwise, where `down(up(R))` is `R`. The return type of `cc` ensures that `cc(r)`s` is `bottom` when `r`s` is `bottom` and the `cc(r)`s` satisfies the invariants spelled out above when it is different from `bottom`. The termination of `cc`, a significant step in the proof, is established by showing that the number of equivalence classes of uninterpreted terms in the domain of `r`s` decreases with each recursive call. The invariants on the solution set play a crucial role in proving termination.

```
cc(r): RECURSIVE {s : above_tinvariants(r`n) | bottom?(r`s)          13
                                          IMPLIES bottom?(s`s)} =
 CASES r`s OF
  bottom: tbottom,
  up(T) : IF (NOT congruence_closed(T))
           THEN cc(merge(r`n, apply(T)(choose(congruent_pair?(T))), T))
           ELSE r
          ENDIF
 ENDCASES
MEASURE cc_rank(r)
```

The `assert(r, aa)` operation places `aa` in canonical form as `aa'`, then expands `r`s` (if `r`s` is `up(T)`) with dummy identities for the new subterms in `aa'` as

`expand(T, aa')`. It then merges `aa'` into this expanded solution set and applies congruence-closure `cc` to the result.

```
assert((r:{r:tinvariants | up?(r`s) IMPLIES                          14
                           congruence_closed(down(r`s))}),
       (aa:typed_equality(r`n))):
   {s:above_tinvariants(r`n) | up?(s`s) IMPLIES
                               congruence_closed(down(s`s))} =
  CASES r`s OF
    bottom: tbottom,
    up(T): cc(merge(r`n, can(T)(aa), expand(T, can(T)(aa))))
  ENDCASES
```

Finally, `process(m, S)` returns a record consisting of a number `n` and a well-typed solution in `n` which may be `bottom`. The type of `process(m, S)` ensures that any solution returned is congruence-closed.

```
process(m, (S:typed_eqlist(m))): RECURSIVE                           15
  {r:above_tinvariants(m) | up?(r`s) IMPLIES
                            congruence_closed(down(r`s))} =
 CASES S OF
  null       : (# n := m, s := up(null)#),
  cons(aa, T): IF up?(process(m, T)`s)
               THEN assert(process(m, T), aa)
               ELSE tbottom
               ENDIF
 ENDCASES
MEASURE length(S)
```

The type and termination proof obligations generated by the PVS typechecker corresponding to the subtype constraints and measures given with the definitions of `process`, `cc`, and other related definitions, ensure the well-typedness and termination of `process`.

4 Verifying Shostak's Algorithm in PVS

The algorithm verifies a sequent $T \vdash a = b$ by computing $S = \mathit{process}(T)$. The sequent is considered valid if either $S = \bot$ or $\mathit{can}(S)(a) \equiv \mathit{can}(S)(b)$. For the soundness of the procedure is established relative to a proof system whose inference rules characterize when a sequent $T \vdash a = b$ is derivable. We prove that the following are equivalent:

1. If $\mathit{process}(T) = S$, then $S = \bot$ or $\mathit{can}(S)(a) \equiv \mathit{can}(S)(b)$.
2. $T \vdash a = b$ is derivable.
3. $T \vdash a = b$ is σ-valid, i.e., valid in all σ-models.

The implication from (1) to (2) is the soundness argument. The implication from (2) to (3) validates the soundness of the proof system with respect to

σ-models. The implication from (3) to (1) establishes the completeness of the decision procedure.

For verifying soundness, we first formally define the class of provable sequents by means of an inductive definition of a predicate `has_proof?(m, T, aa)` for an index `m`, a list of equalities `T`, and an equality `aa`.

16
```
has_proof?(m,
           (T:typed_eqlist(m)),
           (aa:typed_equality(m))): INDUCTIVE bool =
  member(aa, T) OR                                          % Axiom
  lhs(aa) = rhs(aa) OR                                      % Reflexivity
  has_proof?(m, T, eq(rhs(aa), lhs(aa))) OR                 % Symmetry
  (EXISTS (a:typed(m)):                                     % Transitivity
    has_proof?(m, T, eq(lhs(aa), a)) AND
    has_proof?(m, T, eq(a, rhs(aa)))) OR
  (LET a = lhs(aa), b = rhs(aa) IN                          % Congruency
    app?(a) AND app?(b) AND
    func(a) = func(b) AND
    (FORALL (i:arity(func(a))):
      has_proof?(m, T, eq(args(a)(i), args(b)(i))))) OR
  (rhs(aa) = sig(lhs(aa))) OR                               % Canonization
  (EXISTS (bb:typed_equality(m)),                           % Solve
          (n:upfrom(m)), (S:typed_eqlist(n)):
    solve(m, bb, S) AND
    has_proof?(m, T, bb) AND
    has_proof?(n, append(T, S), aa)) OR
  (EXISTS (bb:typed_equality(m)):                           % Contradiction
    unsatisfiable(bb) AND
    has_proof?(m, T, bb))
```

The proof soundness theorem below captures the implication from (2) to (3) above. It asserts that any provable sequent is σ-valid since the variable `M` is declared to range over σ-models. It can be proved by the induction scheme generated by the inductive definition of `has_proof?`.

17
```
proof_soundness: LEMMA
  (FORALL m, (T:typed_eqlist(m)), (aa:typed_equality(m)):
    has_proof?(m, T, aa) IMPLIES
      (FORALL M, (rho:assign(M)): satisfies(M, rho)(T, aa)))
```

The following two theorems correspond to the implication between (1) and (2) above. These theorems capture the respective cases of soundness when `process(m, S)` returns a valid solution or a `bottom` value.

18
```
soundness_1: THEOREM
  (FORALL m, (S:typed_eqlist(m)), (a, b:typed(m)):
    up?(process(m, S)`s) AND
    can(down(process(m, S)`s))(a) = can(down(process(m, S)`s))(b)
   IMPLIES has_proof?(m, S, eq(a, b)))
```

```
soundness_2: THEOREM                                                    19
  (FORALL m, (S:typed_eqlist(m)), (aa:typed_equality(m)):
    bottom?(process(m, S)`s) IMPLIES
      has_proof?(m, S, aa))
```

Completeness is proved by constructing a canonical σ-model M_R and assignment ρ_R, where $R = \mathit{process}(T) \neq \bot$. The bulk of the proof involves showing that this construction does in fact yield a σ-model satisfying the equalities in T. A crucial property for demonstrating this is confluence which asserts that `can(S)(a) = norm(S)(a)` when `S` is congruence-closed and the uninterpreted terms of `a` are included in `dom(S)`.

```
confluence: LEMMA                                                       20
    invariants(S) AND
    congruence_closed(S) AND
    subset?(U(subterm(a)), dom(S)) IMPLIES
      can(S)(a) = norm(S)(a)
```

Completeness is then proved as the theorem below which formalizes the implication from (2) to (1) above, but it is verified via proof soundness and (3). The theorem states that when the sequent $S \vdash a = b$ is derivable, then either $\mathit{process}(S) = \bot$ or $\mathit{process}(S) = T$ and $\mathit{can}(T)(a) = \mathit{can}(T)(b)$.

```
completeness: LEMMA                                                     21
  (FORALL m, (S:typed_eqlist(m)), T, (aa:typed_equality(m)):
    up?(process(m, S)`s) AND
    down(process(m, S)`s) = T AND
    has_proof?(m, S, aa) IMPLIES
      can(T)(lhs(aa)) = can(T)(rhs(aa)))
```

5 Concluding Observations

Both the formalization and the verification closely follow the informal presentation **RS** [RS01]. There were some areas where **RS** was found to be inadequate or incorrect and where PVS itself was deficient.[6]

RS is terse about the introduction of fresh variables by the *solve* operation. These variables must be fresh with respect to the entire execution of the algorithm or the construction of a proof. Proof transformations like weakening and cut require the variables generated by *solve* to be invariant with respect to a certain kind of renaming.[7] The bookkeeping involved in tracking the well-formedness of terms and equalities up to a given index, occupy a substantial

[6] One minor problem was already noticed prior to this verification attempt. Several of the lemmas in the informal proof regarding the composition of solutions were qualified with the condition that $R \cup S$ be functional, where the appropriate condition is that $R \circ S$ must be functional. This was immaterial for the verification since the definition of composition is in terms of lists and not sets.

[7] A similar renaming problem arises with alpha-renaming in the lambda-calculus and *eigenvariables* in sequent proofs, but the renaming issue is far more complicated

fraction of the effort in both the formalization and proof. PVS has a judgement mechanism that records certain typing relations for use in the typechecker, but we were unable to use it for demonstrating that an expression well-typed in n is also well-typed in any index above n.

Quantification over types, needed to define semantic validity, is not expressible in PVS. We instead restricted the semantic domains to subtypes of the type of terms since any model for terms and equalities is essentially characterized by a partition of the term universe into equivalence classes.

A monotonicity lemma is stated in the informal proof (Lemma 3.12) as: *If* $R \cup S$ *is functional, then if* $R(a) \equiv R(b)$*, then* $(R \circ S)(a) \equiv (R \circ S)(b)$*, for any* a *and* b*.* In addition to the above-mentioned correction to the antecedent, this lemma only holds when a and b are in $dom(R)$. Fortunately, only the weak form of this lemma is actually used.

In the **RS** proof of Lemma 5.11, it is claimed that *it can also be shown that* $can(S'^{+})(a) \equiv can(S')(a)$*, and similarly for* b*.* This claim asserts that padding the solution set S' with reflexivity equalities on the subterms from $can(S')(a)$, does not affect the value of $can(S')(a)$. The claim is in fact valid, but the proof is not all that obvious.

Despite the flaws identified above, the **RS** proofs held up quite well to the rigors of formal scrutiny. We were actually operating from a draft document that contained proofs of lemmas that were given without proof in the published version. Once the formalization challenges were overcome, it was possible to make steady progress in the mechanical verification of the proofs. The procedure as we have defined it is not executable since it uses a choice operator. Further work is needed to derive efficiently executable versions of the verified algorithm while preserving its correctness.

The formalization and proof occupied four months of work with PVS carried out entirely by the first author.[8] The proof involves 68 theories, 120 definitions, 192 TCCs (typing and termination proof obligations), 594 lemmas, and the proof checking time is 2,265 seconds on a 1-Gigahertz Pentium 3. There are roughly 6,200 tokens in the detailed informal presentation as measured by a word count of the text file generated from the LaTeX input. There are approximately 13,000 tokens in the PVS specification, and over 25,000 tokens in the PVS proofs. The proof is highly interactive. We are currently working on improving the degree of mechanization in various ways. The level of effort indicates that the certification of complex decision procedures remains a tough challenge.

References

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, NY, 1979.

here. The variable indices affect the type and the well-typedness of equalities and proofs so that renaming is not a local operation.

[8] The first author already had prior experience with PVS having used it for two substantial proof developments[FM01b,FM01a].

[BM81] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[CLS96] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[FM01a] J. Ford and I. A. Mason. Establishing a General Context Lemma in PVS. In *Proceedings of the 2nd Australasian Workshop on Computational Logic, AWCL'01* , 2001. submitted.

[FM01b] J. Ford and I. A. Mason. Operational techniques in PVS—a preliminary evaluation. In *Proceedings of the Australasian Theory Symposium, CATS '01*, Gold Coast, Queensland, Australia, January–February 2001.

[FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification, CAV '2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249, Paris, France, July 2001. Springer-Verlag.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[RS01] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, Boston, MA, July 2001. IEEE Computer Society.

[Sha85] N. Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.

[Sha99] N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1999. Available at `http://www.csl.sri.com/shankar/PVSeval.ps.gz`.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[Thé98] Laurent Théry. A certified version of Buchberger's algorithm. In H. Kirchner and C. Kirchner, editors, *Proceedings of CADE-15*, number 1421 in Lecture Notes in Artificial Intelligence, pages 349–364, Berlin, Germany, July 1998. Springer-Verlag.

[VGL00] Kumar Neeraj Verma and Jean Goubault-Larrecq. Reflecting BDDs in Coq. Technical Report 3859, INRIA, Rocquencourt, France, January 2000.

[vHPPR98] Friedrich W. von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case studies in meta-level theorem proving. In Jim Grundy and Malcolm Newey, editors, *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics*, number 1479 in Lecture Notes in Computer Science, pages 461–478. Springer-Verlag, September 1998.

A Introduction to PVS

We give a very brief introduction to the PVS language and proof checker. PVS specifications are a collection of theories. A theory can have type or individual parameters that are instantiated when the theory is imported within another theory. A parameterized theory can include constraining assumptions on the parameters. The instances of these assumptions corresponding to the actual parameters are generated as proof obligations when a theory instance is imported.

A theory is a list of declarations of types, constants, and formulas. The expression language of PVS is based on simply typed higher-order logic extended with predicate subtypes, dependent types, and recursive datatypes. PVS types consist of the *base* types `bool` and `real`, and *compound* types constructed as tuples, as in [`bool`, `real`], records, as in [#`flag` : `bool`, `length` : `real`#], or function types of the form [$A \rightarrow B$]. Predicates over a type A are of type [$A \rightarrow$`bool`].

Predicate subtypes are a distinctive feature of the PVS higher-order logic. Given a predicate p over A, $\{x : A \mid p(x)\}$ (or, (p)) is a predicate subtype of A consisting of those elements of A satisfying p. The type `nzreal` of nonzero real can be defined as {`x` : `real` | `x /= 0`}. The type `nat` of natural numbers is a predicate subtype of the type `int` of integers, which in turn is a subtype of the subtype `rat` (of `real`) of rational numbers. Subranges can also be defined as predicate subtypes, and arrays can be typed as functions with subranges as domains, e.g., [`below`(`N`)$\rightarrow A$]. The PVS typechecker generates proof obligations (called TCCs) corresponding to predicate subtype constraints. Out-of-bounds array accesses generate unprovable TCCs.

Dependent versions of tuple, record, and function types can be constructed by introducing dependencies between different components of the type through predicates. Dependent typing can be used to define a finite sequence (of arbitrary length) as a dependent record consisting of a length and an array of the given length [#`length` : `nat`, `seq` : [`below`(`length`)$\rightarrow T$]#].

PVS expressions include variables x, constants c, applications $f(a)$, and abstractions `LAMBDA` $(x : T) :\ a$, conditionals `IF` a_1 `THEN` a_2 `ELSE` a_3 `ENDIF`, tuple expressions $(a_1, \ldots, a_n)$, tuple projections $a`i$, record expressions $(\#l_1 :=\! a_1, \ldots \#)$, record projections $a`l$, and (tuple, record, and function) updates $e[a := v]$.

The definition of a recursive datatype can be illustrated with the list type built from the constructors `cons` and `null`. Theories containing the relevant axioms, induction schemes, and useful datatype operations are generated from the datatype declaration.

```
list [T: TYPE]: DATATYPE                                          1
 BEGIN
  null: null?
  cons (car: T, cdr:list):cons?
 END list
```

Invited paper for FLoC'02. Appears in the Proceedings of RTA'02, LNCS.

Combining Shostak Theories*

Natarajan Shankar and Harald Rueß

SRI International Computer Science Laboratory
Menlo Park CA 94025 USA
{shankar, ruess}@csl.sri.com
URL: http://www.csl.sri.com/{~shankar, ~ruess}
Phone: +1 (650) 859-5272 Fax: +1 (650) 859-2844

Abstract. Ground decision procedures for combinations of theories are used in many systems for automated deduction. There are two basic paradigms for combining decision procedures. The Nelson–Oppen method combines decision procedures for disjoint theories by exchanging equality information on the shared variables. In Shostak's method, the combination of the theory of pure equality with canonizable and solvable theories is decided through an extension of congruence closure that yields a canonizer for the combined theory. Shostak's original presentation, and others that followed it, contained serious errors which were corrected for the basic procedure by the present authors. Shostak also claimed that it was possible to combine canonizers and solvers for disjoint theories. This claim is easily verifiable for canonizers, but is unsubstantiated for the case of solvers. We show how our earlier procedure can be extended to combine multiple disjoint canonizable, solvable theories within the Shostak framework.

1 Introduction

Consider the sequent

$$2 * car(x) - 3 * cdr(x) = f(cdr(x)) \\ \vdash f(cons(4 * car(x) - 2 * f(cdr(x)), y)) = f(cons(6 * cdr(x), y)).$$

* This work was funded by NSF Grant CCR-0082560, DARPA/AFRL Contract F33615-00-C-3043, and NASA Contract NAS1-00079. During a phone conversation with the first author on 2nd April 2001, Rob Shostak suggested that the problem of combining Shostak solvers could be solved through variable abstraction. His suggestion is the key inspiration for the combination of Shostak theories presented here. We thank Clark Barrett, Sam Owre, and Ashish Tiwari for their meticulous reading of earlier drafts. We also thank Harald Ganzinger for pointing out certain limitations of our original definition of solvability with respect to σ-models. The first author is grateful to the program committees and program chairs of the FME, LICS, and RTA conferences at FLoC 2002 for their kind invitation.

It involves symbols from three different theories. The symbol f is uninterpreted, the operations $*$ and $-$ are from the theory of linear arithmetic, and the pairing and projection operations *cons*, *car*, and *cdr*, are from the theory of lists. There are two basic methods for building combined decision procedures for disjoint theories, i.e., theories that share no function symbols. Nelson and Oppen [NO79] gave a method for combining decision procedures through the use of variable abstraction for replacing subterms with variables, and the exchange of equality information on the shared variables. Thus, with respect to the example above, decision procedures for pure equality, linear arithmetic, and the theory of lists can be composed into a decision procedure for the combined theory. The other combination method, due to Shostak, yields a decision procedure for the combination of canonizable and solvable theories, based on the congruence closure procedure. Shostak's original algorithm and proof were seriously flawed. His algorithm is neither terminating nor complete (even when terminating). These flaws went unnoticed for a long time even though the method was widely used, implemented, and studied [CLS96,BDL96,Bjø99]. In earlier work [RS01], we described a correct algorithm for the *basic* combination of a single canonizable, solvable theory with the theory of equality over uninterpreted terms. That correctness proof has been mechanically verified using PVS [FS02]. The generality of the basic combination rests on Shostak's claim that it is possible to combine solvers and canonizers from disjoint theories into a single canonizer and solver. This claim is easily verifiable for canonizers, but fails for the case of solvers. In this paper, we extend our earlier decision procedure to the combination of uninterpreted equality with multiple canonizable, solvable theories. The decision procedure does not require the combination of solvers. We present proofs for the termination, soundness, and completeness of our procedure.

2 Preliminaries

We introduce some of the basic terminology needed to understand Shostak-style decision procedures. Fixing a countable set of variables X and a set of function symbols F, a term is either a variable x from X or an n-ary function symbol f from F applied to n terms as in $f(a_1, \ldots, a_n)$. Equations between terms are represented as $a = b$. Let $vars(a)$, $vars(a = b)$, and $vars(T)$ represent the sets of variables in a, $a = b$, and the set of equalities T, respectively. We are interested in deciding the validity of sequents of the form $T \vdash c = d$ where c and d are terms, and T is a set of equalities such that $vars(c = d) \subseteq vars(T)$. The condition $vars(c = d) \subseteq vars(T)$ is there for technical reasons. It can always be satisfied by padding T with reflexivity assertions $x = x$ for any variables x in $vars(c = d) - vars(T)$. We write $[\![a]\!]$ for the set of subterms of a, which includes a.

The semantics for a term a, written as $M[\![a]\!]\rho$, is given relative to an interpretation M over a domain D and an assignment ρ. For an n-ary function f, the interpretation $M(f)$ of f in M is a map from D^n to D. For an *uninterpreted*

n-ary function symbol f, the interpretation $M(f)$ may be any map from D^n to D, whereas only restricted interpretations might be suitable for an interpreted function symbol like the arithmetic + operation. An assignment ρ is a map from variables in X to values in D. We define $M[\![a]\!]\rho$ to return a value in D by means of the following equations.

$$M[\![x]\!]\rho = \rho(x)$$
$$M[\![f(a_1, \ldots, a_n)]\!]\rho = M(f)(M[\![a_1]\!]\rho, \ldots, M[\![a_n]\!]\rho)$$

We say that $M, \rho \models a = b$ iff $M[\![a]\!]\rho = M[\![b]\!]\rho$, and $M \models a = b$ iff $M, \rho \models a = b$ for all assignments ρ. We write $M, \rho \models S$ when $\forall a, b : a = b \in S \Rightarrow M, \rho \models a = b$, and $M, \rho \models (T \vdash a = b)$ when $(M, \rho \models T) \Rightarrow (M, \rho \models a = b)$. A sequent $T \vdash c = d$ is valid, written as $\models (T \vdash c = d)$, when $M, \rho \models (T \vdash c = d)$, for all M and ρ.

There is a simple pattern underlying the class of decision procedures studied here. Let ψ be the state of the decision procedure as given by a set of formulas.[1] Let τ be a family of state transformations so that we write $\psi \xrightarrow{\tau} \psi'$ if ψ' is the result of applying a transformation in τ to ψ, where $vars(\psi) \subseteq vars(\psi')$ (variable preservation). An assignment ρ' is said to extend ρ over $vars(\psi') - vars(\psi)$ when it agrees with ρ on all variables except those in $vars(\psi') - vars(\psi)$ for $vars(\psi) \subseteq vars(\psi')$. We say that ψ' *preserves* ψ if $vars(\psi) \subseteq vars(\psi')$ and for all interpretations M and assignments ρ, $M, \rho \models \psi$ holds iff there exists an assignment ρ' extending ρ such that $M, \rho' \models \psi'$.[2] When preservation is restricted to a limited class of interpretations ι, we say that ψ' ι-preserves ψ. Note that the *preserves* relation is transitive. When the operation τ is deterministic, $\tau(\psi)$ represents the result of the transformation, and we call τ a *conservative* operation to indicate that $\tau(\psi)$ preserves ψ for all ψ. Correspondingly, τ is said to be ι-conservative when $\tau(\psi)$ ι-preserves ψ. Let τ^n represent the n-fold iteration of τ, then τ^n is a conservative operation. The composition $\tau_2 \circ \tau_1$ of conservative operations τ_1 and τ_2, is also a conservative operation. The operation $\tau^*(\psi)$ is defined as $\tau^i(\psi)$ for the least i such that $\tau^{i+1}(\psi) = \tau^i(\psi)$. The existence of such a bound i must be demonstrated for the termination of τ^*. If τ is conservative, so is τ^*.

If τ is a conservative operation, it is sound and complete in the sense that for a formula ϕ with $vars(\phi) \subseteq vars(\psi)$, $\models (\psi \vdash \phi)$ iff $\models (\tau(\psi) \vdash \phi)$. This is clear since τ is a conservative operation and $vars(\phi) \subseteq vars(\psi)$.

[1] In our case, the state is actually represented by a list whose elements are sets of equalities. We abuse notation by viewing such a state as the set of equalities corresponding to the union of the sets of equalities contained in it.

[2] In general, one could allow the interpretation M to be extended to M' in the transformation from ψ to ψ' to allow for the introduction of new function symbols, e.g., skolem functions. This abstract design pattern then also covers skolemization in addition to methods like prenexing, clausification, resolution, variable abstraction, and Knuth-Bendix completion.

If $\tau^*(\psi)$ returns a state ψ' such that $\models (\psi' \vdash \bot)$, where $\bot$ is an unsatisfiable formula, then ψ' and ψ are both clearly unsatisfiable. Otherwise, if ψ' is *canonical*, as explained below, $\models (\psi' \vdash \phi)$ can be decided by computing a canonical form $\psi'[\![\phi]\!]$ for ϕ with respect to ψ'.

3 Congruence Closure

In this section, we present a warm-up exercise for deciding equality over terms where all function symbols are uninterpreted, i.e., the interpretation of these operations is unconstrained. This means that a sequent $T \vdash c = d$ is valid, i.e., $\models (T \vdash c = d)$ iff for all interpretations M and assignments ρ, the satisfaction relation $M, \rho \models (T \vdash c = d)$ holds. Whenever we write $f(a_1, \ldots, a_n)$, the function symbol f is uninterpreted, and $f(a_1, \ldots, a_n)$ is then said to be uninterpreted. Later on, we will extend the procedure to allow interpreted function symbols from disjoint Shostak theories such as linear arithmetic and lists. The congruence closure procedure sets up the template for the extended procedure in Section 5.

The congruence closure decision procedure for *pure equality* has been studied by Kozen [Koz77], Shostak [Sho78], Nelson and Oppen [NO80], Downey, Sethi, and Tarjan [DST80], and, more recently, by Kapur [Kap97]. We present the congruence closure algorithm in a Shostak-style, i.e., as an online algorithm for computing and using canonical forms by successively processing the input equations from the set T. For ease of presentation, we make use of variable abstraction in the style of the abstract congruence closure technique due to Bachmair, Tiwari, and Vigneron [BTV02]. Terms of the form $f(a_1, \ldots, a_n)$ are variable-abstracted into the form $f(x_1, \ldots, x_n)$ where the variables $x_1, \ldots, x_n$ abstract the terms $a_1, \ldots, a_n$, respectively. The procedure shown here can be seen as a specific strategy for applying the abstract congruence closure rules. In Section 5, we make essential use of variable abstraction in the Nelson–Oppen style where it is not merely a presentation device.

Let $T = \{a_1 = b_1, \ldots, a_n = b_n\}$ for $n \geq 0$ so that T is empty when $n = 0$. Let x and y be metavariables that range over variables. The state of the algorithm consists of a *solution state* S and the input equalities T. The solution state S will be maintained as the pair $(S_V; S_U)$, where $(l_1; l_2; \ldots; l_n)$ represents a list with n elements and semi-colon is an associative separator for list elements. The set S_U then contains equalities of the form $x = f(x_1, \ldots, x_n)$ for an n-ary uninterpreted function f, and the set S_V contains equalities of the form $x = y$ between variables. We blur the distinction between the equality $a = b$ and the singleton set $\{a = b\}$. Syntactic identity is written as $a \equiv b$ as opposed to semantic equality $a = b$.

A set of equalities R is *functional* if $b \equiv c$ whenever $a = b \in R$ and $a = c \in R$, for any a, b, and c. If R is functional, it can be used as a lookup table for obtaining the right-hand side entry corresponding to a left-hand side expression. Thus $R(a) = b$ if $a = b \in R$, and otherwise, $R(a) = a$. The domain of R, $dom(R)$

is defined as $\{a \mid a = b \in R \text{ for some } b\}$. When R is not necessarily functional, we use $R(\{a\})$ to represent the set $\{b \mid a = b \in R \vee b \equiv a\}$ which is the image of $\{a\}$ with respect to the reflexive closure of R. The inverse of R, written as R^{-1}, is the set $\{b = a \mid a = b \in R\}$. A functional set R of equalities can be applied as in $R[a]$.

$$\begin{aligned} R[x] &= R(x) \\ R[f(a_1, \ldots, a_n)] &= R(f(R[a_1], \ldots, R[a_n])) \\ R[\{a_1 = b_1, \ldots, a_n = b_n\}] &= \{R[a_1] = R[b_1], \ldots, R[a_n] = R[b_n]\} \end{aligned}$$

In typical usage, R will be a *solution set* where the left-hand sides are all variables, so that $R[a]$ is just the result of applying R as a substitution to a.

When S_V is functional, then S given by $(S_V; S_U)$ can also be used to compute the canonical form $S[\![a]\!]$ of a term a with respect to S. Hilbert's epsilon operator is used in the form of the *when* operator: $F(\overline{x})$ when $\overline{x} : P(\overline{x})$ is an abbreviation for $F(\epsilon\overline{x} : \ P(\overline{x}))$, if $\exists\overline{x} : P(\overline{x})$.

$$\begin{aligned} S[\![x]\!] &= S_V(x) \\ S[\![f(a_1, \ldots, a_n)]\!] &= S_V(x), \text{ when } x : x = f(S[\![a_1]\!], \ldots, S[\![a_n]\!]) \in S_U \\ S[\![f(a_1, \ldots, a_n)]\!] &= f(S[\![a_1]\!], \ldots, S[\![a_n]\!]), \text{ otherwise.} \end{aligned}$$

The set S_V of variable equalities will be maintained so that $vars(S_V) \cup vars(S_U) = dom(S_V)$. The set S_V partitions the variables in $dom(S_V)$ into equivalence classes. Two variables x and y are said to be in the same equivalence class with respect to S_V if $S_V(x) \equiv S_V(y)$. If R and R' are solution sets and R' is functional, then $R \triangleright R' = \{a = R'[b] \mid a = b \in R\}$, and $R \circ R' = R' \cup (R \triangleright R')$. The set S_V is maintained in idempotent form so that $S_V \circ S_V = S_V$. Note that S_U need not be functional since it can, for example, simultaneously contain the equations $x = f(y)$, $x = f(z)$, and $x = g(y)$.

We assume a strict total ordering $x \prec y$ on variables. The operation $orient(x = y)$ returns $\{x = y\}$ if $x \prec y$, and returns $\{y = x\}$, otherwise. The solution state S is said to be *congruence-closed* if $S_U(\{x\}) \cap S_U(\{y\}) = \emptyset$ whenever $S_V(x) \not\equiv S_V(y)$. A solution set S is *canonical* if S is congruence-closed, S_V is functional and idempotent, and S_U is normalized, i.e., $S_U \triangleright S_V = S_U$.

In order to determine if $\models (T \vdash c = d)$, we check if $S'[\![c]\!] \equiv S'[\![d]\!]$ for $S' = process(S; T)$, where $S = (S_V; S_U)$, $S_V = id_T$, $id_T = \{x = x \mid x \in vars(T)\}$, and $S_U = \emptyset$. The congruence closure procedure *process* is defined in Figure 1.

Explanation. We explain the congruence closure procedure using the validity of the sequent $f(f(f(x))) = x,\ x = f(f(x)) \vdash f(x) = x$ as an example. Its validity will be verified by constructing a solution state S' equal to $process(S_V; S_U; T)$ for $T = \{f(f(f(x))) = x, x = f(f(x))\}$, $S_V = id_T$, $S_U = \emptyset$, and checking $S'[\![f(x)]\!] \equiv S'[\![x]\!]$. Note that id_T is $\{x = x\}$. In processing $f(f(f(x))) = x$ with respect to S, the canonization step, $S[\![f(f(f(x))) = x]\!]$

$$
\begin{aligned}
process(S;\emptyset) &= S \\
process(S;\{a=b\}\cup T) &= process(S';T),\ \text{where,} \\
&\quad S' = close^*(merge(abstract^*(S;S[\![a=b]\!]))).
\end{aligned}
$$

$$
\begin{aligned}
close(S) &= merge(S;S_V(x)=S_V(y)), \\
&\quad \text{when } x,y : S_V(x)\not\equiv S_V(y), (S_U(\{x\})\cap S_U(\{y\})\neq\emptyset) \\
close(S) &= S,\ \text{otherwise.}
\end{aligned}
$$

$$
\begin{aligned}
merge(S;x=x) &= S \\
merge(S;x=y) &= (S'_V;\ S'_U),\ \text{where}\ x\not\equiv y, R = orient(x=y), \\
&\qquad\qquad S'_V = S_V\circ R, S'_U = S_U \triangleright R.
\end{aligned}
$$

$$
\begin{aligned}
abstract(S;x=y) &= (S;\ x=y) \\
abstract(S;a=b) &= (S';\ a'=b'), \text{when}\ S', a', b', x_1,\ldots,x_n : \\
&\qquad\qquad f(x_1,\ldots,x_n)\in[\![a=b]\!] \\
&\qquad\qquad x\notin vars(S;a=b) \\
&\qquad\qquad R=\{x=f(x_1,\ldots,x_n)\}, \\
&\qquad\qquad S'=(S_V\cup\{x=x\};\ S_U\cup R), \\
&\qquad\qquad a'=R^{-1}[a], b'=R^{-1}[b].
\end{aligned}
$$

Fig. 1. Congruence closure

yields $f(f(f(x))) = x$, unchanged. Next, the variable abstraction step computes $abstract^*(f(f(f(x))) = x)$. First $f(x)$ is abstracted to v_1 yielding the state $\{x = x, v_1 = v_1\};\{v_1 = f(x)\};\{f(f(v_1)) = x\}$. Variable abstraction eventually terminates renaming $f(v_1)$ to v_2 and $f(v_2)$ to v_3 so that S is $\{x = x, v_1 = v_1, v_2 = v_2, v_3 = v_3\};\{v_1 = f(x), v_2 = f(v_1), v_3 = f(v_2)\}$. The variable abstracted input equality is then $v_3 = x$. Let $orient(v_3 = x)$ return $v_3 = x$. Next, $merge(S; v_3 = x)$ yields the solution state $\{x = x, v_1 = v_1, v_2 = v_2, v_3 = x\};\{v_1 = f(x), v_2 = f(v_1), v_3 = f(v_2)\}$. The congruence closure step $close^*(S)$ leaves S unchanged since there are no variables that are merged in S_U and not in S_V.

The next input equality $x = f(f(x))$ is canonized as $x = v_2$ which can be oriented as $v_2 = x$ and merged with S to yield the new value $\{x = x, v_1 = v_1, v_2 = x, v_3 = x\};\{v_1 = f(x), v_2 = f(v_1), v_3 = f(x)\}$ for S. The congruence closure step $close^*(S)$ now detects that v_1 and v_3 are merged in S_U but not in S_V and generates the equality $v_1 = v_3$. This equality is merged to yield the new value of S as $\{x = x, v_1 = x, v_2 = x, v_3 = x\};\{v_1 = f(x), v_2 = f(x), v_3 = f(x)\}$, which is congruence-closed.

With respect to this final value of the solution state S, it can be checked that $S[\![f(x)]\!] \equiv x \equiv S[\![x]\!]$.

Invariants. The Shostak-style congruence closure algorithm makes heavy use of canonical forms and this requires some key invariants to be preserved on the solution state S. If $vars(S_V) \cup vars(S_U) \subseteq dom(S_V)$, then $vars(S'_V) \cup vars(S'_U) \subseteq dom(S'_V)$, when S' is either $abstract(S; a = b)$ or $close(S)$. If S is canonical and $a' = S[\![a]\!]$, then $S_V[a'] = a'$. If $S_U \triangleright S_V = S_U$, $S_V[a] = a$, and $S_V[b] = b$, then $S'_U \triangleright S'_V = S'_U$ where $S'; a' = b'$ is $abstract(S; a = b)$. Similarly, if $S_U \triangleright S_V = S_U$, $S_V(x) \equiv x$, $S_V(y) \equiv y$, then $S'_U \circ S'_V = S'_U$ for $S' = merge(S; x = y)$. If S_V is functional and idempotent, then so is S'_V, where S' is either of $abstract(S; a = b)$ or $close(S)$. If $S' = close^*(S)$, then S' is congruence-closed, and if S_V is functional and idempotent, S_U is normalized, then S' is canonical.

Variations. In the *merge* operation, if S'_U is computed as $R[S_U]$ instead of $S_U \triangleright R$, this would preserve the invariant that S_U^{-1} is always functional and $S_V[S_U] = S_U$. If this is the case, the canonizer can be simplified to just return $S_U^{-1}(f(S[\![a_1]\!], \ldots, S[\![a_n]\!]))$.

Termination. The procedure $process(S;\ T)$ terminates after each equality in T has been asserted into S. The operation $abstract^*$ terminates because each recursive call decreases the number of occurrences of function applications in the given equality $a = b$ by at least one. The operation $close^*$ terminates because each invocation of the *merge* operation merges two distinct equivalence classes of variables in S_V. The *process* operation terminates because the number of input equations in T decreases with each recursive call. Therefore the computation of $process(S;\ T)$ terminates returning a canonical solution set S'.

Soundness and Completeness. We need to show that $\models (T \vdash c = d) \iff S'[\![c]\!] \equiv S'[\![d]\!]$ for $S' = process(id_T;\ \emptyset;\ T)$ and $vars(c = d) \subseteq vars(T)$. We do this by showing that S' preserves $(id_T;\ \emptyset;\ T)$, and hence $\models (T \vdash c = d) \iff \models (S' \vdash c = d)$, and $\models (S' \vdash c = d) \iff S'[\![c]\!] \equiv S'[\![d]\!]$. We can easily establish that if $process(S;\ T) = S'$, then S' preserves $(S;\ T)$. If $a' = b'$ is obtained from $a = b$ by applying equality replacements from S, then $(S;\ a' = b')$ preserves $(S;\ a = b)$. In particular, $\models (S \vdash S[\![c]\!] = c)$ holds. The following claims can then be easily verified.

1. $(S;\ S[\![a = b]\!])$ preserves $(S;\ a = b)$.
2. $abstract(S;\ a = b)$ preserves $(S;\ a = b)$.
3. $merge(S;\ a = b)$ preserves $(S;\ a = b)$.
4. $close(S)$ preserves S.

The only remaining step is to show that if S' is canonical, then $\models (S' \vdash c = d) \iff S'[\![c]\!] \equiv S'[\![d]\!]$ for $vars(c = d) \subseteq vars(S)$. Since we know that $\models S' \vdash S'[\![c]\!] = c$ and $\models S' \vdash S'[\![d]\!] = d$, hence $\models (S' \vdash c = d)$ follows from $S'[\![c]\!] \equiv S'[\![d]\!]$. For the *only if* direction, we show that if $S'[\![c]\!] \not\equiv S'[\![d]\!]$, then there is an interpretation $M_{S'}$ and assignment $\rho_{S'}$ such that $M_{S'}, \rho_{S'} \models S$ but $M_{S'}, \rho_{S'} \not\models c = d$. A *canonical* term (in S') is a term a such that $S'[\![a]\!] \equiv a$. The domain $D_{S'}$ is taken to be the set of canonical terms built from the function symbols F and variables from $vars(S')$. We constrain $M_{S'}$ so that $M_{S'}(f)(a_1, \ldots, a_n) = S'_V(x)$

when there is an x such that $x = f(a_1, \ldots, a_n) \in S'_U$, and $f(a_1, \ldots, a_n)$, otherwise. Let $\rho_{S'}$ map x in $vars(S')$ to $S'_V(x)$; the mappings for the variables outside $vars(S')$ are irrelevant. It is easy to see that $M_{S'}[\![c]\!]\rho_{S'} = S'[\![c]\!]$ by induction on the structure of c. In particular, when S' is canonical, $M_{S'}(f)(x_1 \ldots, x_n) = x$ for $x = f(x_1, \ldots, x_n) \in S'_U$, so that one can easily verify that $M_{S'}, \rho_{S'} \models S'$. Hence, if $S'[\![c]\!] \not\equiv S'[\![d]\!]$, then $\not\models (S' \vdash c = d)$.

4 Shostak Theories

A Shostak theory [Sho84] is a theory that is canonizable and solvable. We assume a collection of Shostak theories $\theta_1, \ldots, \theta_N$. In this section, we give a decision procedure for a single Shostak theory θ_i, but with i as a parameter. This background material is adapted from Shankar [Sha01]. Satisfiability $M, \rho \models a = b$ is with respect to i-models M. The equality $a = b$ is i-valid, i.e., $\models_i a = b$, if for all i-models M and assignments ρ, $M[\![a]\!]\rho = M[\![b]\!]\rho$. Similarly, $a = b$ is i-unsatisfiable, i.e., $\models_i a \neq b$, when for all i-models M and assignments ρ, $M[\![a]\!]\rho \neq M[\![b]\!]\rho$. An i-term a is a term whose function symbols all belong to θ_i and $vars(a) \subseteq X \cup X_i$.

A canonizable theory θ_i admits a computable operation σ_i on terms such that $\models_i a = b$ iff $\sigma_i(a) \equiv \sigma_i(b)$, for i-terms a and b. An i-term a is canonical if $\sigma_i(a) \equiv a$. Additionally, $vars(\sigma_i(a)) \subseteq vars(a)$ and every subterm of $\sigma_i(a)$ must be canonical. For example, a canonizer for the theory θ_A of linear arithmetic can be defined to convert expressions into an ordered sum-of-monomials form. Then, $\sigma_A(y + x + x) \equiv 2 * x + y \equiv \sigma_A(x + y + x)$.

A solvable theory admits a procedure $solve_i$ on equalities such that $solve_i(Y)(a = b)$ for a set of variables Y with $vars(a = b) \subseteq Y$, returns a solved form for $a = b$ as explained below. $solve_i(Y)(a = b)$ might contain fresh variables that do not appear in Y. A functional solution set R is in *i-solved form* if it is of the form $\{x_1 = t_1, \ldots, x_n = t_n\}$, where for j, $1 \leq j \leq n$, t_j is a canonical i-term, $\sigma_i(t_j) \equiv t_j$, and $vars(t_j) \cap dom(R) = \emptyset$ unless $t_j \equiv x_j$. The i-solved form $solve_i(Y)(a = b)$ is either $\bot_i$, when $\models_i a \neq b$, or is a solution set of equalities which is the union of sets R_1 and R_2. The set R_1 is the solved form $\{x_1 = t_1, \ldots, x_n = t_n\}$ with $x_j \in vars(a = b)$ for $1 \leq j \leq n$, and for any i-model M and assignment ρ, we have that $M, \rho \models a = b$ iff there is a ρ' extending ρ over $vars(solve_i(Y)(a = b)) - Y$ such that $M, \rho' \models x_j = t_j$, for $1 \leq j \leq n$. The set R_2 is just $\{x = x \mid x \in vars(R_1) - Y\}$ and is included in order to preserve variables. In other words, $solve_i(Y)(a = b)$ i-preserves $a = b$. For example, a solver for linear arithmetic can be constructed to isolate a variable on one side of the equality through scaling and cancellation. We assume that the fresh variables generated by $solve_i$ are from the set X_i. We take $vars(\bot_i)$ to be $X \cup X_i$ so as to maintain variable preservation, and indeed $\bot_i$ could be represented as just $\bot$ were it not for this condition.

We now describe a decision procedure for sequents of the form $T \vdash c = d$ in a single Shostak theory with canonizer σ_i and solver $solve_i$. Here the solution state

S is just a functional solution set of equalities in i-solved form. Given a solution set S, we define $S\langle\langle a\rangle\rangle_i$ as $\sigma_i(S[a])$. The composition of solutions sets is defined so that $S \circ_i \bot_i = \bot_i \circ_i S = \bot_i$ and $S \circ_i R = R \cup \{a = R\langle\langle b\rangle\rangle_i \mid a = b \in S\}$. Note that solved forms are idempotent with respect to composition so that $S \circ_i S = S$. The solved form $solveclose_i(id_T;\ T)$ is obtained by processing the equations in T to build up a solution set S. An equation $a = b$ is first canonized with respect to S as $S\langle\langle a\rangle\rangle_i = S\langle\langle b\rangle\rangle_i$ and then solved to yield the solution R. If R is $\bot_i$, then T is i-unsatisfiable and we return the solution state with $S_i = \bot_i$ as the result. Otherwise, the composition $S \circ_i R$ is computed and used to similarly process the remaining formulas in T.

$$
\begin{aligned}
solveclose_i(S;\ \emptyset) &= S \\
solveclose_i(\bot_i;\ T) &= \bot_i \\
solveclose_i(S;\ \{a = b\} \cup T) &= solveclose_i(S', T), \\
&\qquad \text{where } S' = S \circ_i solve_i(vars(S))(S\langle\langle a\rangle\rangle_i = S\langle\langle b\rangle\rangle_i)
\end{aligned}
$$

To check i-validity, $\models_i (T \vdash c = d)$, it is sufficient to check that either $solveclose_i(id_T;\ T) = \bot$ or $S'\langle\langle c\rangle\rangle_i \equiv S'\langle\langle d\rangle\rangle_i$, where $S' = solveclose_i(id_T;\ T)$.

Soundness and Completeness. As with the congruence closure procedure, each step in $solveclose_i$ is i-conservative. Hence $solveclose_i$ is sound and complete: if $S' = solveclose_i(S;\ T)$, then for every i-model M and assignment ρ, $M, \rho \models S \cup T$ iff there is a ρ' extending ρ over the variables in $vars(S') - vars(S)$ such that $M, \rho' \models S'$. If $\sigma_i(S'[a]) \equiv \sigma_i(S'[b])$, then $M, \rho' \models a = S'[a] = \sigma_i(S'[a]) = \sigma_i(S'[b]) = S'[b] = b$, and hence $M, \rho \models a = b$. Otherwise, when $\sigma_i(S'[a]) \not\equiv \sigma_i(S'[b])$, we know by the condition on σ_i that there is an i-model M and an assignment ρ' such that $M[\![S'[a]]\!]\rho' \neq M[\![S'[b]]\!]\rho'$. The solved form S' divides the variables into independent variables x such that $S'(x) = x$, and dependent variables y where $y \neq S'(y)$ and the variables in $vars(S'(y))$ are all independent. We can therefore extend ρ' to an assignment ρ where the dependent variables y are mapped to $M[\![S'(y)]\!]\rho'$. Clearly, $M, \rho \models S'$, $M, \rho \models a = S'[a]$, and $M, \rho \models b = S'[b]$. Since S' i-preserves $(id_T;\ T)$, $M, \rho \models T$ but $M, \rho \not\models a = b$ and hence $T \vdash a = b$ is not i-valid, so the procedure is complete. The correctness argument is thus similar to that of Section 3 but for the case of a single Shostak theory considered here, there is no need to construct a canonical term model since $\models_i a = \sigma_i(a)$, and $\sigma_i(a) \equiv \sigma_i(b)$ iff $\models_i a = b$.

Canonical term model. The situation is different when we wish to combine Shostak theories. It is important to resolve potential semantic incompatibilities between two Shostak theories. With respect to some fixed notion of i-validity for θ_i and j-validity for θ_j with $i \neq j$, a formula A in the union of θ_i and θ_j may be satisfiable in an i-interpretation of only a specific finite cardinality for which there might be no corresponding satisfying j-interpretation for the formula. Such an incompatibility can arise even when a theory θ_i is extended with uninterpreted function symbols. For example, if ϕ is a formula with variables x and y that is satisfiable only in a two-element model M where $\rho(x) \neq \rho(y)$, then

the set of formulas Γ where $\Gamma = \{\phi, f(x) = x, f(u) = y, f(y) = x\}$ additionally requires $\rho(x) \neq \rho(u)$ and $\rho(y) \neq \rho(u)$. Hence, a model for Γ must have at least three elements, so that Γ is unsatisfiable. However, there is no way to detect this kind of unsatisfiability purely through the use of solving and canonization.

We introduce a canonical term model as a way around such semantic incompatibilities. The set of canonical i-terms a such that $\sigma_i(a) \equiv a$ yields a domain for a *term model* M_i where $M_i(f)(a_1, \ldots, a_n) = \sigma_i(f(a_1, \ldots, a_n))$. If M_i is (isomorphic to) an i-model, then we say that the theory θ_i is *composable*. Note that the *solve* operation is conservative with respect to the model M_i as well, since M_i is taken as an i-model.

Given the usual interpretation of disjunction, a notion of validity is said to be *convex* when $\models (T \vdash c_1 = d_1 \vee \ldots \vee c_n = d_n)$ implies $\models (T \vdash c_k = d_k)$ for some k, $1 \leq k \leq n$. If a theory θ_i is composable, then i-validity is convex. Recall that $\models_i (T \vdash c_1 = d_1 \vee \ldots \vee c_n = d_n)$ iff $\models_i (S \vdash c_1 = d_1 \vee \ldots \vee c_n = d_n)$ for $S = \mathit{solveclose}_i(id_T; T)$. If $S = \bot_i$, then $\models_i (T \vdash c_k = d_k)$, for $1 \leq k \leq n$. If $S \neq \bot_i$, then since S i-preserves T, $\models_i (S \vdash c_1 = d_1 \vee \ldots \vee c_n = d_n)$, but (by assumption) $\not\models_i (S \vdash c_k = d_k)$. An assignment ρ_S can be constructed so that for independent (i.e., where $S(x) = x$) variables $x \in \mathit{vars}(S)$, $\rho_S(x) = x$, and for dependent variables $y \in \mathit{vars}(S)$, $\rho_S(y) = M_i[\![S(y)]\!]\rho_S$. If for $S \neq \bot_i$, $\not\models_{\sigma_i} (S \vdash c_k = d_k)$, then $M_i, \rho_S \models S$ and $M_i, \rho_S \not\models c_k = d_k$. Hence $M_i, \rho_S \not\models (S \vdash c_k = d_k)$, for $1 \leq k \leq n$. This yields $M_i, \rho_S \not\models (T \vdash c_1 = d_1 \vee \ldots \vee c_n = d_n)$, contradicting the assumption.

5 Combining Shostak Theories

We now examine the combination of the theory of equality over uninterpreted function symbols with several disjoint Shostak theories. Examples of interpreted operations from Shostak theories include $+$ and $-$ from the theory of linear arithmetic, *select* and *update* from the theory of arrays, and *cons*, *car*, and *cdr* from the theory of lists. The basic Shostak combination algorithm covers the union of equality over uninterpreted function symbols and a single canonizable and solvable equational theory [Sho84,CLS96,RS01]. Shostak [Sho84] had claimed that the basic combination algorithm was sufficient because canonizers and solvers for disjoint theories could be combined into a single canonizer and solver for their union. This claim is incorrect.[3] We present a combined decision procedure for multiple Shostak theories that overcomes the difficulty of combining solvers.

Two theories θ_1 and θ_2 are said to be disjoint if they have no function symbols in common. A typical subgoal in a proof can involve interpreted symbols from several theories. Let σ_i be the canonizer for θ_i. A term $f(a_1, \ldots, a_n)$ is said to be in θ_i if f is in θ_i even though some a_i might contain function symbols outside θ_i. In processing terms from the union of pairwise disjoint theories $\theta_1, \ldots, \theta_N$,

[3] The difficulty with combining Shostak solvers was observed by Jeremy Levitt [Lev99].

it is quite easy to combine the canonizers so that each theory treats terms in the other theory as variables. Since σ_i is only applicable to i-terms, we first have to extend the canonizer σ_i to treat terms in θ_j for $j \neq i$, as variables. Let π_i be a chosen bijective set of equalities between the variables X and the set $\{a|(\exists j : j \neq i \wedge a \in \theta_j)\}$. We treat uninterpreted function symbols as belonging to a special theory θ_0 where $\sigma_0(a) = a$ for $a \in \theta_0$. The extended operation σ_i' is defined below.

$$\sigma_i'(a) = \pi_i[\sigma_i(a')], \text{ when } a' : \begin{array}{l} a' \text{ is an } i\text{-term}, \\ \pi_i[a'] \equiv a. \end{array}$$

Note that the *when* condition in the above definition can always be satisfied. The combined canonizer σ can then be defined as

$$\begin{aligned} \sigma(x) &= x \\ \sigma(f(a_1, \ldots, a_n)) &= \sigma_i'(f(\sigma(a_1), \ldots, \sigma(a_n))), \text{ when } i : f \text{ is in } \theta_i. \end{aligned}$$

This canonizer is, however, not used in the remainder of the paper.

We now discuss the difficulty of combining the solvers $solve_1$ and $solve_2$ for θ_1 and θ_2, respectively, into a single solver. The example uses the theory θ_A of linear arithmetic and the theory θ_L of the pairing and projection operations *cons*, *car*, *cdr*, where, somewhat nonsensically, the projection operations also apply to numerical expressions. Shostak illustrated the combination using the example

$$5 + car(x + 2) = cdr(x + 1) + 3.$$

Since the top-level operation on the left-hand side is $+$, we can treat $car(x+2)$ and $cdr(x+1)$ as variables and use $solve_A$. This might yield a partially solved equation of the form $car(x + 2) = cdr(x + 1) - 2$. Now since the top-level operation on the left-hand side is from the theory of lists, we use $solve_L$ to obtain $x + 2 = cons(cdr(x + 1) - 2, u)$ with a fresh variable u. We once again apply $solve_A$ to obtain $x = cons(cdr(x + 1) - 2, u) - 2$. This is, however, not in solved form: the left-hand side variable occurs in an interpreted context in its solution. There is no way to prevent this from happening as long as each solver treats terms from another theory as variables. Therefore the union of Shostak theories is not necessarily a Shostak theory.

The problem of combining disjoint Shostak theories actually has a very simple solution. There is no need to combine solvers. Since the theories are disjoint, the canonizer can tolerate multiple solutions for the same variable as long as there is at most one solution from any individual theory. This can be illustrated on the same example: $5 + car(x + 2) = cdr(x + 1) + 3$. By variable abstraction, we obtain the equation $v_3 = v_6$, where $v_1 = x + 2, v_2 = car(v_1), v_3 = v_2 + 5, v_4 = x + 1, v_5 = cdr(v_4), v_6 = v_5 + 3$. We can separate these equations out into the respective theories so that S is $(S_V;\ S_U;\ S_A;\ S_L)$, where S_V contains the variable equalities in canonical form, S_U is as in congruence closure but is always $\emptyset$ since there are no uninterpreted operations in this example, and S_A and S_L are the

solution sets for θ_A and θ_L, respectively. We then get $S_V = \{x = x, v_1 = v_1, v_2 = v_2, v_3 = v_6, v_4 = v_4, v_5 = v_5, v_6 = v_6\}$, $S_A = \{v_1 = x + 2, v_3 = v_2 + 5, v_4 = x + 1, v_6 = v_5 + 3\}$, and $S_L = \{v_2 = car(v_1), v_5 = cdr(v_4)\}$. Since v_3 and v_6 are merged in S_V, but not in S_A, we solve the equality between $S_A(v_3)$ and $S_A(v_6)$, i.e., $solve_A(v_2 + 5 = v_5 + 3)$ to get $v_2 = v_5 - 2$. This result is composed with S_A to get $\{v_1 = x + 2, v_3 = v_5 + 3, v_4 = x + 1, v_6 = v_5 + 3, v_2 = v_5 - 2\}$ for S_A. There are no new variable equalities to be propagated out of either S_A, S_L, or S_V. Notice that v_2 and v_5 both have different solved forms in S_A and S_L. This is tolerated since the solutions are from disjoint theories and the canonizer can pick a solution that is appropriate to the context. For example, when canonizing a term of the form $f(x)$ for $f \in \theta_i$, it is clear that the only relevant solution for x is the one from S_i.

We can now check whether the resulting solution state verifies the original equation $5 + car(x + 2) = cdr(x + 1) + 3$. In canonizing $f(a_1, \ldots, a_n)$ we return $S_V(y)$ whenever the term $f(S_i(S[\![a_1]\!]), \ldots, S_i(S[\![a_n]\!]))$ being canonized is such that $y = f(S_i(S[\![a_1]\!]), \ldots, S_i(S[\![a_n]\!])) \in S_i$ for $f \in \theta_i$. Thus $x+2$ canonizes to v_1 using S_A, and $car(v_1)$ canonizes to v_2 using S_L. The resulting term $5+v_2$, using the solution for v_2 from S_A, simplifies to $v_5 + 3$, which returns the canonical form v_6 by using S_A. On the right-hand side, $x + 1$ is equivalent to v_4 in S_A, and $car(v_4)$ simplifies to v_5 using S_L The right-hand side therefore simplifies to v_5+3 which is canonized to v_6 using S_A. The canonized left-hand and right-hand sides are identical.

We present a formal description of the procedure used informally in the above example. We show how *process* from Section 3 can be extended to combine the union of disjoint solvable, canonizable, composable theories. We assume that there are N disjoint theories $\theta_1, \ldots, \theta_N$. Each theory θ_i is equipped with a canonizer σ_i and solver $solve_i$ for i-terms. If we let I represent the interval $[1, N]$, then an I-model is a model M that is an i-model for each $i \in I$. We will ensure that each inference step is conservative with respect to I-models, i.e., I-conservative. We represent the uninterpreted part of S as S_0 instead of S_U. The solution state S of the algorithm now consists of a list of sets of equations $(S_V;\ S_0;\ S_1;\ \ldots;\ S_N)$. Here S_V is a set of variable equations of the form $x = y$, and S_0 is the set of equations of the form $x = f(x_1, \ldots, x_n)$ where f is uninterpreted. Each S_i is in i-solved form and is the solution set for θ_i.

Terms now contain a mixture of function symbols that are uninterpreted or are interpreted in one of the theories θ_i. A solution state S is *confluent* if for all $x, y \in dom(S_V)$ and i, $0 \leq i \leq N$: $S_V(x) \equiv S_V(y) \iff S_i(\{x\}) \cap S_i(\{y\}) \neq \emptyset$. A solution state S is canonical if it is confluent; S_V is functional and idempotent, i.e., $S_V \circ S_V = S_V$; the uninterpreted solution set S_0 is normalized, i.e., $S_0 \triangleright S_V = S_0$; each S_i, for $i > 0$, is functional, idempotent, i.e., $S_i \circ_i S_i = S_i$, normalized i.e., $S_i \triangleright S_V = S_i$, and in i-solved form. The canonization of expressions with respect to a canonical solution set S is defined as follows.

$$S[\![x]\!] \stackrel{\triangle}{=} S_V(x)$$

$$
\begin{aligned}
&\mathit{abstract}(S;\ x = y) = (S;\ x = y),\\
&\mathit{abstract}(S;\ a = b) = (S';\ a' = b'),\\
&\qquad\qquad \text{when } S', c, i:\ c \in \mathit{max}(\lceil a = b \rceil_i),\\
&\qquad\qquad\qquad\qquad x \notin \mathit{vars}(S \cup a = b),\\
&\qquad\qquad\qquad\qquad S'_V = S_V \cup \{x = x\},\\
&\qquad\qquad\qquad\qquad S'_i = S_i \cup \{x = c\},\\
&\qquad\qquad\qquad\qquad S'_j = S_j, \text{ for }, i \neq j\\
&\qquad\qquad\qquad\qquad a' = S'[\![a]\!],\\
&\qquad\qquad\qquad\qquad b' = S'[\![b]\!].
\end{aligned}
$$

Fig. 2. Variable abstraction step for multiple Shostak theories

$$
\begin{aligned}
S[\![f(a_1,\ldots,a_n)]\!] &\stackrel{\triangle}{=} S_V(x), \text{ when } i, x :\\
&\qquad i > 0, f \in \theta_i, x = S\{\{f(a_1,\ldots,a_n)\}\} \in S_i\\
S[\![f(a_1,\ldots,a_n)]\!] &\stackrel{\triangle}{=} S\{\{f(a_1,\ldots,a_n)\}\}, \text{ otherwise.}
\end{aligned}
$$

$$
\begin{aligned}
S\{\{f(a_1,\ldots,a_n)\}\} &\stackrel{\triangle}{=} \sigma'_i(f(S_i(S[\![a_1]\!]),\ldots,S_i(S[\![a_n]\!]))),\\
&\qquad \text{if } f \in \theta_i, i > 0\\
S\{\{f(a_1,\ldots,a_n)\}\} &\stackrel{\triangle}{=} f(S[\![a_1]\!],\ldots,S[\![a_n]\!]), \text{ if } f \in \theta_0.
\end{aligned}
$$

Since variables are used to communicate between the different theories, the canonical variable x in S_V is returned when the term being canonized is known to be equivalent to an expression a such that $y = a$ in S_i, where $x \equiv S_V(y)$. The definition of the above global canonizer is one of the key contributions of this paper. This definition can be applied to the example above of computing $S[\![5 + \mathit{car}(x + 2)]\!]$.

Variable Abstraction. The variable abstraction procedure $\mathit{abstract}(S; a = b)$ is shown in Figure 2. If a is an i-term such that $a \notin X$, then a is said to be a pure i-term. Let $\lceil a = b \rceil_i$ represent the set of subterms of $a = b$ that are pure i-terms. A maximal 0-term is one of the form $f(x_1,\ldots,x_n)$ for $f \in \theta_0$. For $i > 0$, the set $\mathit{max}(M)$ of maximal terms in M is defined to be $\{a \in M | a \equiv b \vee a \notin \lceil b \rceil$, for any $b \in M\}$. In a single variable abstraction step, $\mathit{abstract}(S;\ a = b)$ picks a maximal pure i-subterm c from the canonized input equality $a = b$, and replaces it with a fresh variable x from X while adding $x = c$ to S_i. By abstracting a maximal pure i-term, we ensure that S_i remains in i-solved form.

Explanation. The procedure in Figure 3 is similar to that of Figure 1. Equations from the input set T are processed into the solution state S of the form $S_V; S_0; \ldots, S_N$. Initially, S must be canonical. In processing the input equation $a = b$ into S, we take steps to systematically restore the canonicity of S. The first step is to compute the canonical form $S[\![a = b]\!]$ of $a = b$ with respect to S. It is easy to see that $(S; S[\![a = b]\!])$ I-preserves $(S; a = b)$.

$$
\begin{aligned}
process(S;\ \emptyset) &= S \\
process(S;\ T) &= S,\ \text{when } i : S_i = \bot_i \\
process(S;\ \{a = b\} \cup T) &= process(S';\ T),\ \text{where} \\
&\qquad S' = close^*(merge_V(abstract^*(S;\ S[\![a = b]\!]))).
\end{aligned}
$$

$$
\begin{aligned}
close(S) &= S,\ \text{when } i : S_i = \bot_i \\
close(S) &= S',\ \text{when } S', i, x, y : \\
&\qquad x, y \in dom(S_V), \\
&\qquad (i > 0, S_V(x) \equiv S_V(y), S_i(x) \not\equiv S_i(y),\ \text{and} \\
&\qquad\ \ S' = merge_i(S;\ x = y)) \\
&\qquad \text{or} \\
&\qquad (i \geq 0, S_V(x) \not\equiv S_V(y), S_i(\{x\}) \cap S_i(\{y\}) \neq \emptyset,\ \text{and} \\
&\qquad\ \ S' = merge_V(S;\ S_V(x) = S_V(y))) \\
close(S) &= normalize(S),\ \text{otherwise.}
\end{aligned}
$$

$$
normalize(S) = (S_V;\ S_0;\ S_1 \triangleright S_V;\ \ldots;\ S_N \triangleright S_V).
$$

$$
\begin{aligned}
merge_i(S; x = y) &= S',\ \text{where } i > 0, \\
&\qquad S'_i = S_i \circ_i solve_i(vars(S_i))(S_i(x) = S_i(y)), \\
&\qquad S'_j = S_j,\ \text{for } i \neq j, \\
&\qquad S'_V = S_V. \\
merge_V(S; x = x) &= S \\
merge_V(S; x = y) &= (S_V \circ R;\ S_0 \triangleright R;\ S_1;\ \ldots;\ S_N),\ \text{where } R = orient(x = y).
\end{aligned}
$$

Fig. 3. Combining Multiple Shostak Theories

The result of the canonization step $a' = b'$ is then variable abstracted as $abstract^*(a' = b')$ (shown in Figure 2) so that in each step, a maximal, pure i-subterm c of $a' = b'$ is replaced by a fresh variable x, and the equality $x = c$ is added to S_i. This is also easily seen to be an I-conservative step. The equality $x = y$ resulting from the variable abstraction of $a' = b'$ is then merged into S_V and S_0. This can destroy confluence since there may be variables w and z such that w and z are merged in S_V (i.e., $S_V(w) \equiv S_V(z)$) that are unmerged in some S_i (i.e., $S_i(\{w\}) \cap S_i(\{z\}) = \emptyset$), or vice-versa.[4] The number of variables in $dom(S_V)$ remains fixed during the computation of $close^*(S)$. Confluence is restored by $close^*(S)$ which finds a pair of variables that are merged in some S_i but not in S_V, and merging them in S_V, or that are merged in S_V and not in some S_i and merging them in S_i. Each such merge step is also I-conservative. When this process terminates, S is once again canonical. The solution sets S_i are normalized with respect to S_V in order to ensure that the entries are in the normalized form for lookup during canonization.

[4] For $i > 0$, S_i is maintained in i-solved form and hence, $S_i(\{x\}) = \{x, S_i(x)\}$.

Invariants. As with congruence closure, several key invariants are needed to ensure that the solution state S is maintained in canonical form whenever it is given as the argument to *process*. If S is canonical and a and b are canonical with respect to S, then for $(S'; a' = b') = \mathit{abstract}(S;\ a = b)$, S' is canonical, and a' and b' are canonical with respect to S'. The state $\mathit{abstract}(S;\ a = b)$ I-preserves $(S;\ a = b)$. A solution state is said to be well-formed if S_V is functional and idempotent, S_0 is normalized, and each S_i is functional, idempotent, and in solved form. Note that if S is well-formed, confluent, and each S_i is normalized, then it is canonical. When S is well-formed, and $S' = \mathit{merge}_V(S; x = y)$ or $S' = \mathit{merge}_i(S; x = y)$, then S' is well-formed and I-preserves $(S; x = y)$. If S is well-formed and congruence-closed, and $S' = \mathit{normalize}(S)$, then S' is well-formed and each S'_i is normalized. If $S' = \mathit{normalize}(S)$, then each S'_i is in solved form because if x replaces y on the right-hand side of a solution set S_i, then $S_i(y) \equiv y$ since S_i is in i-solved form. By congruence closure, we already have that $S_i(x) \equiv S_i(y) \equiv y$. Therefore, the uniform replacement of y by x ensures that $S'_i(x) \equiv x$, thus leaving S in solved form. If $S' = \mathit{close}^*(S)$, where S is well-formed, then S' is canonical.

Variations. As with congruence closure, once S is confluent, it is safe to strengthen the normalization step to replace each S_i by $S_V[S_i]$. This renders S_0^{-1} functional, but S_i^{-1} may still be non-functional for $i > 0$, since it might contain left-hand side variables that are local. However, if $\hat{S}_i$ is taken to be S_i restricted to $\mathit{dom}(S_V)$, then $\hat{S}_i^{-1}$ with the strengthened normalization is functional and can be used in canonization. The solutions for local variables can be safely discarded in an actual implementation. The canonization and variable abstraction steps can be combined within a single recursion.

Termination. The operations $S[\![a = b]\!]$ and $\mathit{abstract}^*(S; a = b)$ are easily seen to be terminating. The operation $\mathit{close}^*(S)$ also terminates because the sum of the number of equivalence classes of variables in $\mathit{dom}(S_V)$ with respect to each of the solution sets $S_V, S_0, S_1, \ldots, S_N$, decreases with each *merge* operation.

Soundness and Completeness. We have already seen that each of the steps: canonization, variable abstraction, composition, merging, and normalization, is I-conservative. It therefore follows that if $S' = \mathit{process}(S; T)$, then S' I-preserves S. Hence, if $S'[\![c]\!] \equiv S'[\![d]\!]$, then clearly $\models_I (S' \vdash c = d)$, and hence $\models_I (S; T \vdash c = d)$.

The completeness argument requires the demonstration that if $S'[\![c]\!] \not\equiv S'[\![d]\!]$, then $\not\models_I (S' \vdash c = d)$ when S' is canonical. This is done by means of a construction of $M_{S'}$ and $\rho_{S'}$ such that $M_{S'}, \rho_{S'} \models S'$ but $M_{S'}, \rho_{S'} \not\models c = d$. The domain D consists of canonical terms e such that $S'[\![e]\!] = e$. As with congruence closure, $M_{S'}$ is defined so that $M_{S'}(f)(e_1, \ldots, e_n) = S'[\![f(e_1, \ldots, e_n)]\!]$. The assignment $\rho_{S'}$ is defined so that $\rho_{S'}(x) = S_V(x)$. By induction on c, we have that $M_{S'}[\![c]\!]\rho_{S'} = S'[\![c]\!]$. We can also easily check that $M_{S'}, \rho_{S'} \models S'$.

It is also the case that $M_{S'}$ is an I-model since $M_{S'}$ is isomorphic to M_i for each i, $1 \leq i \leq N$. This can be demonstrated by constructing a bijective

map μ_i between D and the domain D_i corresponding to M_i so that $\mu_i(x) = a'$, where $\pi_i[a'] = S_I(x)$, $\mu_i(f(a_1, \ldots, a_n)) = f(\mu_i(a_1), \ldots, \mu_i(a_n))$ if $f \in \theta_i$, and $\pi_i^{-1}(f(a_1, \ldots, a_n)$, otherwise. It can then be verified that for any $f \in \theta_i$ and terms $a_1, \ldots, a_n$ in D, $\mu_i(M_{S'}(f)(a_1, \ldots, a_n)) = M_i(f)(\mu_i(a_1), \ldots, \mu_i(a_n))$. This concludes the proof of completeness.

Convexity revisited. As in Section 4, the term model construction of $M_{S'}$ once again establishes that I-validity is convex. In other words, a sequent $\models_I (T \vdash c_1 = d_1 \vee \ldots \vee c_n = d_n)$ iff $\models_I (T \vdash c_k = d_k)$ for some k, $1 \leq k \leq n$.

6 Conclusions

Ground decision procedures for equality are crucial for discharging the myriad proof obligations that arise in numerous applications of automated reasoning. These goals typically contain operations from a combination of theories, including uninterpreted symbols. Shostak's basic method deals only with the combination of a single canonizable, solvable theory with equality over uninterpreted function symbols. Indeed, in all previous work based on Shostak's method, only the basic combination is considered. Though Shostak asserted that the basic combination was adequate to cover the more general case of multiple Shostak theories, this claim has turned out to be unsubstantiated. We have given here the first Shostak-style combination method for the general case of multiple Shostak theories. The algorithm is quite simple and is supported by straightforward arguments for termination, soundness, and completeness.

Shostak's combination method, as we have described it, is clearly an instance of a Nelson–Oppen combination [NO79] since it involves the exchange of equalities between variables through the solution set S_V. The added advantage of a Shostak combination is that it combines the canonizers of the individual theories into a global canonizer. The definition of such a canonizer for multiple Shostak theories is the key contribution of this paper. The technique of achieving confluence across the different solution sets is unique to our method. Confluence is needed for obtaining useful canonical forms, and is therefore not essential in a general Nelson–Oppen combination. The global canonizer $S[\![a]\!]$ can be applied to input formulas to discharge queries and simplify input formulas. The reduction to canonical form with respect to the given equalities helps keep the size of the term universe small, and makes the algorithm more efficient than a black box Nelson–Oppen combination. The decision algorithm for a Shostak theory given in Section 4 fits the requirements for a black box procedure that can be used within a Nelson–Oppen combination. The Nelson–Oppen combination of Shostak theories with other decision procedures has been studied by Tiwari [Tiw00], Barrett, Dill, and Stump [BDS02], and Ganzinger [Gan02], but none of these methods includes a general canonization procedure as is required for a Shostak combination.

Variable abstraction is also used in the combination unification procedure of Baader and Schulz [BS96], which addresses a similar problem to that of combining Shostak solvers. In our case, there is no need to ensure that solutions are compatible across distinct theories. Furthermore, variable dependencies can be cyclic across theories so that it is possible to have $y \in vars(S_i(x))$ and $x \in vars(S_j(y))$ for $i \neq j$. Our algorithm can be easily and usefully adapted for combining unification and matching algorithms with constraint solving in Shostak theories.

Insights derived from the Nelson–Oppen combination method have been crucial in the design of our algorithm and its proof. Our presentation here is different from that of our previous algorithm for the basic Shostak combination [RS01] in the use of variable abstraction and the theory-wise separation of solution sets. Our proof of the basic algorithm additionally demonstrated the existence of proof objects in a sound and complete proof system. This can easily be replicated for the general algorithm studied here. The soundness and completeness proofs given here are for composable theories and avoid the use of σ-models.

Our Shostak-style algorithm fits modularly within the Nelson–Oppen framework. It can be employed within a Nelson–Oppen combination (as suggested by Rushby [CLS96]) in which there are other decision procedures that generate equalities between variables. It is also possible to combine it with decision procedures that are not disjoint, as for example with linear arithmetic inequalities. Here, the existence of a canonizer with respect to equality is useful for representing inequality information in a canonical form. A variant of the procedure described here is implemented in ICS [FORS01] in exactly such a combination.

References

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201, Palo Alto, CA, November 1996. Springer-Verlag.

[BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, editor, *Frontiers of Combining Systems, 4th International Workshop, FroCos 2002*, number 2309 in Lecture Notes in Artificial Intelligence, pages 132–146, Berlin, Germany, April 2002. Springer-Verlag.

[Bjø99] Nikolaj Bjørner. *Integrating Decision Procedures for Temporal Verification.* PhD thesis, Stanford University, 1999.

[BS96] F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *J. Symbolic Computation*, 21:211–243, 1996.

[BTV02] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 2002. To appear.

[CLS96] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, edi-

tors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[DST80] P.J. Downey, R. Sethi, and R.E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, 1980.

[FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification, CAV '2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249, Paris, France, July 2001. Springer-Verlag.

[FS02] Jonathan Ford and Natarajan Shankar. Formal verification of a combination decision procedure. In A. Voronkov, editor, *Proceedings of CADE-19*, Berlin, Germany, 2002. Springer-Verlag.

[Gan02] Harald Ganzinger. Shostak light. In A. Voronkov, editor, *Proceedings of CADE-19*, Berlin, Germany, 2002. Springer-Verlag.

[Kap97] Deepak Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *International Conference on Rewriting Techniques and Applications, RTA '97*, number 1232 in Lecture Notes in Computer Science, pages 23–37, Berlin, 1997. Springer-Verlag.

[Koz77] Dexter Kozen. Complexity of finitely presented algebras. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, pages 164–177, Boulder, Colorado, 2–4 May 1977.

[Lev99] Jeremy R. Levitt. *Formal Verification Techniques for Digital Systems.* PhD thesis, Stanford University, 1999.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[RS01] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, Boston, MA, July 2001. IEEE Computer Society.

[Sha01] Natarajan Shankar. Using decision procedures with a higher-order logic. In *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 5–26, Edinburgh, Scotland, September 2001. Springer-Verlag. Available at ftp://ftp.csl.sri.com/pub/users/shankar/tphols2001.ps.gz.

[Sho78] R. Shostak. An algorithm for reasoning about equality. *Comm. ACM*, 21:583–585, July 1978.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[Tiw00] Ashish Tiwari. *Decision Procedures in Automated Deduction.* PhD thesis, State University of New York at Stony Brook, 2000.

On the Confluence of Linear Shallow Term Rewrite Systems

Guillem Godoy*, Ashish Tiwari**, and Rakesh Verma***

Technical University of Catalonia	SRI International	Computer Science Dept
Jordi Girona 1, Barcelona, Spain	Menlo Park, CA	Univ of Houston, TX
ggodoy@lsi.upc.es	*tiwari@csl.sri.com*	*rmverma@cs.uh.edu*

Abstract. We show that the confluence of shallow linear term rewrite systems is decidable. The decision procedure is a nontrivial generalization of the polynomial time algorithms for deciding confluence of ground and restricted non-ground term rewrite systems presented in [13, 2]. Our algorithm has a polynomial time complexity if the maximum arity of a function symbol in the signature is considered a constant. We also give EXPTIME-hardness proofs for reachability and confluence of shallow term rewrite systems.

1 Introduction

Programming language interpreters, proving equations (e.g. $x^3 = x$ implies the ring is Abelian), abstract data types, program transformation and optimization, and even computation itself (e.g., Turing machine) can all be specified by a set of rules, called a rewrite system. The rules are used to replace ("reduce") subexpressions of given expressions by other expressions (usually equivalent ones in some sense). A fundamental property of a rewrite system is the confluence or Church-Rosser property. Informally, confluence states that if an expression a can be reduced (in zero or more steps) to two different expressions b and c, then there is a common expression d to which b and c can be reduced in zero or more steps. Confluence implies uniqueness of normal ("irreducible") forms and helps to "determinise" their search by avoiding backtracking.

In general, confluence is well-known to be undecidable; however, it is known to be decidable for terminating systems [8] and for the subclass of arbitrary variable-free ("ground") systems [4, 11]. Ground systems include as a subclass the tree automata model, which has important computer science applications. The previous decidability proofs of confluence for ground systems [4, 11] were based on tree-automata techniques and showed that this problem was in EXPTIME, but no nontrivial lower bounds were known. Hence the exact complexity of this problem was open until last year, when a series of papers [7, 2, 13, 6] culminated in a polynomial time algorithm for this problem for shallow and rule-linear systems, which include ground systems as a special case. In a shallow system variables in the rules cannot appear at depth more than one. Shallow

* Partially supported by the Spanish CICYT project MAVERISH ref. TIC2001-2476-C03-01.
** Research supported in part by DARPA under the MoBIES and SEC programs administered by AFRL under contracts F33615-00-C-1700 and F33615-00-C-3043, and NSF CCR-0082560.
*** Research supported in part by NSF grant CCR-9732186.

systems have been well-studied in other contexts [10, 3]. Linearity in [13, 6], called rule-linearity here, means each variable can appear at most once in the entire rule. Thus, commutativity ($x+y = y+x$) is a shallow equation but *not* rule-linear, and associativity is neither shallow nor rule-linear.

In this paper, we establish decidability of confluence for shallow systems in which the left-hand side and right-hand side are *independently* linear, i.e., a variable can have two occurrences in a rule—once in each side of the rule. In fact, the decision procedure runs in polynomial time if we assume that the maximum arity of a function symbol in the signature is a constant. The results in this paper subsume the shallow rule-linear systems of [13] as a special case. The algorithm is a nontrivial generalization of the algorithms in [2, 13]. We introduce a notion of marked terms and marked rewriting, and then generalize the central concept of top stability in [2]. The conditions to be checked by the algorithm are also generalized and the constructions are more involved. We also prove that the reachability, joinability and confluence problems are all EXPTIME-hard for shallow non-linear systems and all are known to be undecidable for linear non-shallow systems [15], which indicates that the linearity and shallowness assumptions are fairly tight.

1.1 Preliminaries

Let $\mathcal{F}$ be a (finite) set of function symbols with an arity function $arity\colon \mathcal{F} \to I\!N$. Function symbols f with $arity(f) = n$, denoted by $f^{(n)}$, are called *n-ary* symbols (when $n = 1$, one says *unary* and when $n = 2$, *binary*). If $arity(f) = 0$, then f is a *constant symbol*. Let $\mathcal{X}$ be a set of variable symbols. The set of terms over $\mathcal{F}$ and $\mathcal{X}$, denoted by $\mathcal{T}(\mathcal{F}, \mathcal{X})$, is the smallest set containing all constant and variable symbols such that $f(t_1, \ldots, t_n)$ is in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ whenever $f \in \mathcal{F}$, $arity(f) = n$, and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. A *position* is a sequence of positive integers. If p is a position and t is a term, then by $t|_p$ we denote the *subterm of t at position p*: we have $t|_\lambda = t$ (where λ denotes the empty sequence) and $f(t_1, \ldots, t_n)|_{i.p} = t_i|_p$ if $1 \leq i \leq n$ (and is undefined if $i > n$). We also write $t[s]_p$ to denote the term obtained by replacing in t the subterm at position p by the term s. For example, if t is $f(a, g(b, h(c)), d)$, then $t|_{2.2.1} = c$, and $t[d]_{2.2} = f(a, g(b, d), d)$. By $|s|$ we denote the *size* (number of symbols) of a term s: we have $|a| = 1$ if a is a constant symbol or a variable, and $|f(t_1, \ldots, t_n)| = 1 + |t_1| + \ldots + |t_n|$. The *depth* of a term s is 0 if s is a variable or a constant, and $1 + max_i depth(s_i)$ if $s = f(s_1, \ldots, s_m)$. Terms with depth 0 are denoted by α, β, with possible subscripts.

If $\rightarrow$ is a binary relation on a set S, then $\rightarrow^+$ is its transitive closure, $\leftarrow$ is its inverse, and $\rightarrow^*$ is its reflexive-transitive closure. Two elements s and t of S are called *joinable* by $\rightarrow$, denoted $s \downarrow t$, if there exists a u in S such that $s \rightarrow^* u$ and $t \rightarrow^* u$. The relation $\rightarrow$ is called *confluent* or *Church-Rosser* if the relation $\leftarrow^* \circ \rightarrow^*$ is contained in $\rightarrow^* \circ \leftarrow^*$, that is, for all s, t_1 and t_2 in S, if $s \rightarrow^* t_1$ and $s \rightarrow^* t_2$, then $t_1 \downarrow t_2$. An equivalent definition of confluence of $\rightarrow$ is that $\leftrightarrow^*$ is contained in $\rightarrow^* \circ \leftarrow^*$, that is, all s and t in S such that $s \leftrightarrow^* t$ are joinable.

A *substitution* σ is a mapping from variables to terms. It can be homomorphically extended to a function from terms to terms: using a postfix notation, $t\sigma$ denotes the result of simultaneously replacing in t every $x \in Dom(\sigma)$ by $x\sigma$. Substitutions are

sometimes written as finite sets of pairs $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where each x_i is a variable and each t_i is a term. For example, if σ is $\{x \mapsto f(b, y), y \mapsto a\}$, then $g(x, y)\sigma$ is $g(f(b, y), a)$.

A *rewrite rule* is a pair of terms (l, r), denoted by $l \to r$, with left-hand side (lhs) l and right-hand side (rhs) r. A *term rewrite system* (TRS) R is a finite set of rewrite rules. We say that s rewrites to t in one step at position p (by R), denoted by $s \to_{R,p} t$, if $s|_p = l\sigma$ and $t = s[r\sigma]_p$, for some $l \to r \in R$ and substitution σ. If $p = \lambda$, then the rewrite step is said to be applied *at the topmost position* (at the root) and is denoted by $s \to_R^r t$; it is denoted by $s \to_R^{nr} t$ otherwise. The rewrite relation $\to_R$ induced by R on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is defined by $s \to_R t$ if $s \to_{R,p} t$ for some position p.

A *(rewrite) derivation or proof* (from s) is a sequence of rewrite steps (starting from s), that is, a sequence $s \to_R s_1 \to_R s_2 \to_R \ldots$. The *size* $|R|$ of a TRS R of the form $\{l_1 \to r_1, \ldots, l_n \to r_n\}$ is $|l_1| + |r_1| + \ldots + |l_n| + |r_n|$.

Definition 1. *A term t is called*

- linear *if no variable occurs more than once in t.*
- shallow *if no variable occurs in t at depth greater than 1, i.e., if $t|_p$ is a variable, then p is a position of length zero or one.*
- flat *if t is a non-constant term of the form $f(s_1, \ldots, s_n)$ where all s_i are variables or constants.*

Definition 2. *Let R be a TRS.*
A term s is reachable *from t by R if $t \to_R^* s$.*
Two terms s and t are equivalent *by R if $s \leftrightarrow_R^* t$.*
Two terms s and t are joinable *by R, denoted by $s \downarrow_R t$, if they are joinable by $\to_R$.*
*A term s is R-*irreducible *if there is no term t s.t. $s \to_R t$.*
The TRS R is confluent *if the relation $\to_R$ is confluent on $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

We assume that R is a shallow and linear term rewrite system, that is, if $s \to t$ is a rule in R, then s and t are both linear and shallow terms. Unlike previous results in [13, 6], the terms s and t are allowed to share variables.

2 Confluence of Shallow and Linear Rewrite Systems

Assuming that the maximum arity of a function symbol in $\mathcal{F}$ is bounded by a constant, we show that confluence of shallow and linear term rewrite system R over $\mathcal{F}$ can be decided in polynomial time. The proof of this fact uses suitable generalizations of the techniques in [2, 13]. In Section 2.1 we argue that without loss of generality, we can restrict the signature $\mathcal{F}$ to contain exactly one function symbol with nonzero arity. Thereafter, we transform the rewrite system R into a flat rewrite system in Section 2.2. The flat linear term rewrite system is saturated under ordered chaining inference rule in Section 2.3 to construct a rewrite closure, which has several useful properties. The rest of the proof relies on the notion of top-stable and marked top-stable terms (Section 2.4), the ability to compute these sets (Section 2.5), and relating confluence of a saturated flat linear rewrite system to efficiently checkable properties over these sets (Section 2.6).

2.1 Simplifying the Signature

Terms over an arbitrary signature $\mathcal{F}$ can be encoded by terms over a signature $\mathcal{F}'$ containing at most one function symbol with non-zero arity. We may assume that $\mathcal{F}$ contains at least one constant e that does not appear in R.

Proposition 1. *There exists an injective mapping σ from terms over an arbitrary $\mathcal{F}$ to terms over a signature $\mathcal{F}'$ containing exactly one function symbol (with non-zero fixed arity) such that if R' is defined as $\{\sigma(s) \to \sigma(t) : s \to t \in R\}$, then R is confluent if, and only if, R' is confluent.*

Proof. (Sketch) Let m be one plus the maximum arity of any function symbol in $\mathcal{F}$. Define the new signature $\mathcal{F}'$ as

$$\mathcal{F}' = \{h^{(0)} : h^{(l)} \in \mathcal{F}, l > 0\} \cup \{f^{(m)}\} \cup \{c^{(0)} : c^{(0)} \in \mathcal{F}\},$$

where f is a new symbol. Define the map σ as follows: for each $h \in \mathcal{F}$ with arity $l > 0$,

$$\sigma(h(t_1, \ldots, t_l)) = f(\sigma(t_1), \ldots, \sigma(t_l), e, \ldots, e, h)$$

where the number of e's above equals $m - l - 1$, and for each $c \in \mathcal{F}$ with arity 0, $\sigma(c) = c$. The mapping σ is clearly injective, but not surjective. We can classify terms over $\mathcal{F}'$ into type 1 and type 2 terms (using a simple sorted signature) so that terms of type 1 exactly correspond to $Range(\sigma)$. It is easy to see that there is a bijective correspondence between proofs in R and proofs in R' over terms in $Range(\sigma)$. Combining this observation with a result in [16], which states that proving confluence for arbitrary terms over the signature is equivalent to proving confluence of the well-typed terms according to any many-sorted discipline which is compatible with the rewrite system under consideration, it follows that R is confluent iff R' is confluent.

2.2 Flat Representation

In the transformation described in Section 2.1, the properties of being linear and shallow are preserved. We next flatten the term rewrite system so that the depth of each term is at most one. In particular, given a linear shallow term rewrite system R, it can be transformed so that each rule in R is of the form

$f(\alpha_1, \ldots, \alpha_m) \to c$	(F_c)	$c \to f(\alpha_1, \ldots, \alpha_m)$	(B_c)
$f(\alpha_1, \ldots, \alpha_m) \to x$	(F_x)	$x \to f(\alpha_1, \ldots, \alpha_m)$	(B_x)
$f(\alpha_1, \ldots, \alpha_m) \to f(\beta_1, \ldots, \beta_m)$	(P_f)	$\alpha \to \beta$	(P_c)

where each $\alpha_i, \beta_i, \alpha, \beta$ is a depth 0 term (i.e., either a variable or a constant). Rules of the form F_c and F_x are called *forward* rules and denoted by F, rules of the form B_c and B_x are called *backward* rules and denoted by B, and rules of the form P_f and P_c are called *permutation* rules and denoted by P. Rules of the form B_x are called *insertion* rules. We call such a rewrite system R a *flat linear* rewrite system.

This transformation is easily done by replacing each non-constant ground term, say s, in R by a new constant, say c, and adding a rule $s \to c$ or $c \to s$, depending on whether s occurred on the left- or right-hand side of R.

Flatten: $$\frac{u[s] \to t}{u[c] \to t, s \to c} \qquad \frac{t \to u[s]}{t \to u[c], c \to s}$$

where s is a non-constant ground term and c is a new constant.

Exhaustive application of these two rules results in a flat linear shallow rewrite system. This transformation can be done in polynomial time, as the number of applications of the above two rules is bounded by the size of the initial rewrite system R. It is easily seen to preserve confluence, see [2, 13] for instance.

2.3 Rewrite Closure

Let $\succ$ order terms based on their size, that is, $s \succ t$ iff $|s| > |t|$. An application of an F-rule results in a smaller term, whereas application of a B-rule gives a bigger term in this ordering.

Definition 3. *A term s is size-irreducible by R if there exists no term t such that $s \to_R^* t$ and $s \succ t$.*

Definition 4. *A derivation $s \to_R^* t$ is said to be increasing if for all decompositions $s \to_R^* s' \to_{l \to r, p} t' \to_R^* t$, there is no step at a prefix position of p in $t' \to_R^* t$.*

Observe that increasing derivations either have no rewrite step at position λ, or only one at the beginning of the derivation. For simplicity, we eliminate the former case by assuming a dummy rewrite rule $x \to x$ to be in R, which can always be applied at the λ position in case there is no top step.

A flat linear rewrite system can be saturated under the following ordered chaining inference to give an enlarged flat linear rewrite system with some nice properties.

Ordered Chaining: $$\frac{s \to t \qquad w[u] \to v}{w[s]\sigma \to v\sigma} \qquad \frac{s \to w[t] \qquad u \to v}{s\sigma \to w[v]\sigma}$$

where σ is the most general unifier of t and u, neither u nor t is a variable, and $s \not\succ t$ in the first case and $v \not\succ u$ in the second. Note that these restrictions ensure that ordered chaining preserves flatness and shallowness.

Application of ordered chaining preserves confluence. Moreover, if the maximum arity m is a constant, then saturation under ordered chaining can be performed in polynomial time.

Lemma 1. *Let $R = F \cup B \cup P$ be a flat linear rewrite system saturated under the ordered chaining inference rules. If $s \to_R^* t$, then there is a proof of the form $s \to_F^* \circ \to_P^* \circ \to_B^* t$.*

Lemma 1 can be easily established using proof simplification arguments [1]. Similar proofs have been presented before, but for the special case of ground systems [12] and rule-linear shallow rewrite systems [14]. The generalization to linear shallow case is straightforward and the details are skipped here. The process of saturation, in this context, can be interpreted as asymmetric completion [9].

Lemma 2. *Let R be a flat linear rewrite system saturated under the ordered chaining inference rule above. If s is size-irreducible (or, equivalently F-irreducible) and $s \to_R^* t$, then there is an increasing derivation $s \to_R^* t$.*

Example 1. If $R = \{x + y \to y + x, x \to 0 + x\}$, then the chaining inferences add a new rule $x \to x + 0$ to R. An increasing derivation for $0 + x \to^* (x + 0) + 0$ is $0 + x \to x + 0 \to (x + 0) + 0$.

2.4 Top-Stable Terms, Marked Terms, and Marked Rewriting

In the rest of the paper we assume that R is a flat linear term rewrite system, which is also saturated under the chaining inference rule.

Definition 5. *A term t with depth greater than 0 is said to be* top-stable *if it cannot be reduced to a depth 0 term. A depth 0 term α is* top-stabilizable *if it is equivalent to a top-stable term.*

The following is a simple consequence of Lemma 1.

Lemma 3. *The set $S_0 = \{f\alpha_1 \dots \alpha_m : f\alpha_1 \dots \alpha_m$ is F-irreducible$\}$ is the set of all top-stable flat terms.*

The confluence test relies heavily on the concept of top-stable terms and depth 0 top-stabilizable terms. The basic observation is that if a top-stabilizable constant, say c, occurs at a certain position in a term, say $fct_2 \dots t_m$, then this term $(fct_2 \dots t_m)$ is equivalent to a term $t = ft_1t_2 \dots t_m$ with the property that t rewrites to a depth 0 term via R only if $fxt_2 \dots t_m$ also does. Here, t_1 is chosen to be top-stable. So, when considering rewrites on $fct_2 \dots t_m$ or $ft_1 \dots t_m$, we should treat c and t_1 as variables. This is roughly the intuition behind the following definitions of marked terms and marked rewriting.

Definition 6. *A marking M of a term t is a set of leaf positions in t. A term t with a marking M is denoted by (t, M).*

A marked term (s, M) rewrites to (t, N) via marked rewriting if $s \to_{l \to r \in R, p} t$ for some position $p \notin M$ such that, if $l|_{p_1}$ is a constant then $p.p_1 \notin M$, and the new marking N satisfies: (a) for all q disjoint with p, we have $q \in M$ iff $q \in N$, (b) for all p_1, p_2 and q such that $l|_{p_1}$ and $r|_{p_2}$ are the same variable, $p.p_1.q \in M$ iff $p.p_2.q \in N$, and (c) no more positions are in N.

A marked flat term $(s = f\alpha_1 \dots \alpha_m, M)$ is said to be correctly marked if for all $i \in M$, we have that α_i is top-stabilizable.

Example 2. The marked term $(0 + x, \{1\})$, denoted as $\underline{0} + x$, cannot be rewritten with the rule $0 + x \to x$, but it can be rewritten with the rule $x + y \to y + x$ to $x + \underline{0}$.

The notions of size-irreducible terms, increasing derivations and top-stable terms can be adapted naturally to marked terms. All the arguments of Lemmas 1 and 2 are also valid for marked rewriting, and we have:

Lemma 4. *If $(s, M) \to_R^* (t, N)$, then there is a derivation of the form $(s, M) \to_F^* \circ \to_P^* \circ \to_B^* (t, N)$. If (s, M) is size-irreducible and $(s, M) \to_R^* (t, N)$, then there is an increasing derivation $(s, M) \to_R^* (t, N)$.*

2.5 The Sets S_∞ and J_∞

The set S_0 of top-stable flat terms can be extended with empty markings to give the new set

$$\{(f\alpha_1 \ldots \alpha_m, \emptyset) : f\alpha_1 \ldots \alpha_m \in S_0\}$$

of marked top-stable terms, which we also denote by S_0. We add new marked flat terms to this set to get the set of all correctly marked top-stable flat terms and top-stabilizable constants (and a variable if some variable is top-stabilizable) using the following fixpoint computation, starting with the new set S_0.

$$\begin{aligned} S_{j+1} = S_j \cup \{c : c \leftrightarrow^* f\alpha_1 \ldots \alpha_m \text{ for some } (f\alpha_1 \ldots \alpha_m, M) \in S_j\} \\ \cup \{(f\alpha_1 \ldots \alpha_m, M) : (f\alpha_1 \ldots \alpha_m, M) \text{ is top-stable and } \forall i \in M : \alpha_i \in S_j\} \end{aligned}$$

Note that by Lemma 4, $(f\alpha_1 \ldots \alpha_m, M)$ is top-stable iff it is irreducible by F by marked rewriting.

This iterative procedure of computing larger and larger subsets S_j of the set of all marked flat terms is guaranteed to terminate in a polynomial number of steps. This is because the total number of flat marked terms, up to variable renaming, is polynomial, assuming m is a constant.

Lemma 5. *If S_∞ is the fixpoint of the computation above, then, up to variable renaming, $(f\alpha_1 \ldots \alpha_m, M) \in S_\infty$ iff $(f\alpha_1 \ldots \alpha_m, M)$ is top-stable and correctly marked, and a depth 0 term $c \in S_\infty$ iff c is top-stabilizable.*

Definition 7. *Two marked terms (s, M) and (t, N) are said to be structurally joinable if $(s, M) \rightarrow_R^* (s', M')$ and $(t, N) \rightarrow_R^* (t', N')$ for some terms s' and t' with the same structure (i.e., $Pos(s') = Pos(t')$, where $Pos(s')$ is the set of all positions in s') and equivalent leaf terms (i.e., for all leaf positions[1] $p \in Pos(s')$, we have that $s'|_p$ and $t'|_p$ are equivalent).*

We use the following fixpoint computation to obtain some structurally joinable pairs of marked terms.

$$\begin{aligned} J_0 = \{((\alpha, \emptyset), (\beta, M)) : \alpha \leftrightarrow_R^* \beta \text{ and } \alpha, \beta \text{ are depth 0 terms}\} \\ J_{j+1} = J_j \cup \{((\alpha, \emptyset), (f\beta_1 \ldots \beta_m, M)) : \\ (f\beta_1 \ldots \beta_m, M) \text{ is top-stable,} \\ \alpha \rightarrow_R^r fa_1 \ldots a_m, (f\beta_1 \ldots \beta_m, M) \rightarrow_R^r (fb_1 \ldots b_m, N), \text{ and} \\ \forall i \in \{1 \ldots m\} \text{ either } a_i = b_i \text{ or } ((a_i, \emptyset), (b_i, N|_i)) \in J_j\} \end{aligned}$$

where $N|_i$ contains the positions p such that $i.p \in N$. Note that the b_i can be considered depth 0 or 1 terms, and that the a_i can be considered depth 0 or satisfying $a_i = b_i$.

Lemma 6. *If J_∞ is the fixpoint of above computation, then it is the set of all structurally joinable pairs of terms of the form $((\alpha, \emptyset), (\beta, M))$ or $((\alpha, \emptyset), (f\beta_1 \ldots \beta_m, M))$, where α, β are depth 0 terms, and $(f\beta_1 \ldots \beta_m, M)$ is a flat top-stable marked term.*

[1] Note that due to Proposition 1, for non-leaf positions $p \in Pos(s')$, $s'|_p = t'|_p = f$.

2.6 The Technical Lemma and the Result

Definition 8. *A pair of rules* $((l \to r), (l' \to r'))$ *is* useless *if* $l = x$ *and* $l' = y$ *for some variables* x *and* y *that appear in* r *and* r'*, respectively, at the same non-root position.*

Two top-stable marked flat terms $(f\alpha_1 \dots \alpha_m, M)$ *and* $(f\alpha'_1 \dots \alpha'_m, M')$ *are* first-step joinable *if there exist*

$$(f\alpha_1 \dots \alpha_m, M) \to^r_{(l\to r)\sigma} (fs_1 \dots s_m, N)$$
$$(f\alpha'_1 \dots \alpha'_m, M') \to^r_{(l'\to r')\theta} (fs'_1 \dots s'_m, N')$$

such that every s_i *is equivalent to its corresponding* s'_i*, and* $((l \to r), (l' \to r'))$ *is not useless.*

Note that first-step joinability can be efficiently computed, since it is enough to consider subterms s_i and s'_i of depth 0 or 1: if $r|_i$ is a variable not in l, we can force $r|_i\sigma = r'|_i\theta$ by modifying the substitutions, and the same if $r'|_i$ is a variable not in l'.

The polynomial time test for confluence depends on the following characterization using the sets S_∞ and J_∞ and the notion of first-step joinability.

Lemma 7. *The rewrite system* R *is confluent if, and only if,*

(c1) Every pair α, β *of equivalent depth* 0 *terms is joinable,*
(c2) If $\alpha \leftrightarrow^* f\beta_1 \dots \beta_m$ *and* $(f\beta_1 \dots \beta_m, M) \in S_\infty$ *then* $(\alpha, \emptyset)$ *and* $(f\beta_1 \dots \beta_m, M)$ *are structurally joinable, i.e.* $((\alpha, \emptyset), (f\beta_1 \dots \beta_m, M)) \in J_\infty$
(c3) If $(f\alpha_1 \dots \alpha_m, M) \in S_\infty$ *and* $(f\beta_1 \dots \beta_m, N) \in S_\infty$ *are such that* $f\alpha_1 \dots \alpha_m \leftrightarrow^* f\beta_1 \dots \beta_m$*, then these two marked terms in* S_∞ *are first-step joinable.*

Proof. (Sketch) A correctly marked flat term $(f\alpha_1 \dots \alpha_m, M)$ can be *lifted* to a term $fs_1 \dots s_m$ by replacing the marked α_i's by equivalent top-stable and F-irreducible terms s_i's.

$\Rightarrow$: Suppose R is confluent. Condition (c1) follows from the definition of confluence.

Condition (c2). Suppose $\alpha \leftrightarrow^* f\beta_1 \dots \beta_m$ and $(f\beta_1 \dots \beta_m, M) \in S_\infty$. Using Lemma 5, we can lift $(f\beta_1 \dots \beta_m, M)$ to the term $ft_1 \dots t_m$. Since $(f\beta_1 \dots \beta_m, M)$ is top-stable, it follows that $ft_1 \dots t_m$ is top-stable. Now, α is equivalent to $ft_1 \dots t_m$ and by confluence they are joinable. Since both are size-irreducible, there are increasing derivations of the form $\alpha \to^*_R u$ and $ft_1 \dots t_m \to^*_R u$. We can extract an increasing derivation $(f\beta_1 \dots \beta_m, M) \to^*_R (u' = u[\beta'_1]_{p_1} \dots [\beta'_k]_{p_k}, N = \{p_1, \dots, p_k\})$ from the latter derivation by ignoring all rewrite steps at or below marked positions. Using an auxiliary lemma, we can show that there exists a derivation $\alpha \to^*_R u[\beta''_1]_{p_1} \dots [\beta''_m]_{p_k}$, such that $\beta''_i \leftrightarrow^*_R \beta'_i$.

Condition (c3). Let $(f\alpha_1 \dots \alpha_m, M)$ and $(f\beta_1 \dots \beta_m, N)$ be marked flat terms in S_∞ such that $f\alpha_1 \dots \alpha_m \leftrightarrow^*_R f\beta_1 \dots \beta_m$. Again, using Lemma 5, we can lift $(f\alpha_1 \dots \alpha_m, M)$ and $(f\beta_1 \dots \beta_m, N)$ to size-irreducible terms $s = fs_1 \dots s_m$ and $t = ft_1 \dots t_m$. By confluence, s and t are joinable, and hence there exist increasing derivations $fs_1 \dots s_m \to^r_{(l\to r)\sigma} fs'_1 \dots s'_m \to^{*,nr}_R u$ and $ft_1 \dots t_m \to^r_{(l'\to r')\theta} ft'_1 \dots t'_m \to^{*,nr}_R u$. Such a u can be chosen minimally, and consequently the pair $(l \to r, l' \to r')$ is not useless. Clearly, every s'_i is equivalent to the corresponding t'_i.

Now, by suitably modifying the substitutions σ and θ, we can get marked rewrite steps, $(f\alpha_1 \ldots \alpha_m, M) \rightarrow_{(l \rightarrow r)\sigma'} (fs''_1 \ldots s''_m, M')$ and $(f\beta_1 \ldots \beta_m, N) \rightarrow_{(l' \rightarrow r')\theta'} (ft''_1 \ldots t''_m, N')$, such that $s''_i \leftrightarrow^*_R s'_i \leftrightarrow^*_R t'_i \leftrightarrow^*_R t''_i$. This shows that the two marked terms $(f\alpha_1 \ldots \alpha_m, M)$ and $(f\beta_1 \ldots \beta_m, N)$ are first-step joinable.

$\Leftarrow$: Suppose conditions (c1), (c2), and (c3) are satisfied, but R is not confluent. Let $\{s, t\}$ be a witness to non-confluence, that is, s and t are equivalent, but not joinable. We compare witnesses by a multiset extension of the ordering $\succ$ defined earlier. First, we note that both s and t can be assumed to be size-irreducible, otherwise we would have a smaller counterexample to confluence.

If $s = fs_1 \ldots s_m$, then each s_i is either top-stable or of depth 0. Similarly, for the term t. Additionally, if the top-stable subterms s_i are equivalent to some depth 0 terms, then the term s can be *projected* onto a correctly marked flat term $(f\alpha_1 \ldots \alpha_m, M)$ where either α_i is the depth 0 term equivalent to s_i and $i \in M$, or $\alpha_i = s_i$. We differentiate the following cases based on the form of s and t:

Case 1. s and t are both depth 0 terms: In this case, Condition (c1) implies that s and t are joinable, a contradiction.

Case 2. s is a depth 0 term α and $t = ft_1 \ldots t_m$: We first claim that each t_i is equivalent to a depth 0 term. If not, then w.l.o.g. let t_1 not be equivalent to any depth 0 term. Then t_1 cannot be "used" in the proof $\alpha \leftrightarrow^*_R ft_1 \ldots t_m$, and hence it can be replaced by a new variable x in this proof to yield a new proof $\alpha' \leftrightarrow^*_R fxt_2 \ldots t_m$. If α' and $fxt_1 \ldots t_m$ are not joinable, then they are a smaller witness to non-confluence, a contradiction. If α' and $fxt_1 \ldots t_m$ are joinable, then $\alpha' = x$, and α and t_1 are equivalent, but not joinable. The pair $\{\alpha, t_1\}$ is a smaller witness to non-confluence, a contradiction again.

Let $(f\beta_1 \ldots \beta_m, M)$ be a projection of t. This marked term is top-stable and correctly marked, and hence by Lemma 5, it is in S_∞. By condition (c2), $(\alpha, \emptyset)$ and $(f\beta_1 \ldots \beta_m, M)$ are structurally joinable, and hence, there exist $(\alpha, \emptyset) \rightarrow^*_R (s', \emptyset)$ and $(f\beta_1 \ldots \beta_m, M) \rightarrow^*_R (t', M')$ such that $Pos(s') = Pos(t')$, and for every leaf position $p \in Pos(s')$ we have $s'|_p \leftrightarrow^*_R t'|_p$. If $M' = \{p_1 \ldots p_k\}$, then $t' = t'[\beta|_{i_1}]_{p_1} \ldots [\beta|_{i_k}]_{p_k}$, for some $i_1 \ldots i_k \subseteq M$.

If we mimic the derivation $f\beta_1 \ldots \beta_m \rightarrow^*_R t'$, but now starting from $ft_1 \ldots t_m$, we obtain a derivation of the form $ft_1 \ldots t_m \rightarrow^*_R t'' = t'[t_{i_1}]_{p_1} \ldots [t_{i_k}]_{p_k}$. Moreover, each t_{i_j} is equivalent to $t'|_{p_j} = \beta_{i_j}$, and hence, for each leaf position p of s' we have that $s'|_p$ and $t''|_p$ are equivalent, and $size(t''|_p) < size(t)$. Since α and $ft_1 \ldots t_m$ are not joinable, s' and t'' are not joinable, and hence, for some leaf position p of s' we have that $s'|_p$ and $t''|_p$ are not joinable. For such a p, $\{s'|_p, t''|_p\}$ is a smaller witness to non-confluence, a contradiction.

Case 3. $s = fs_1 \ldots s_m$ and $t = ft_1 \ldots t_m$: Using arguments similar to the previous case, we can assume that the s_i's and t_i's are equivalent to depth 0 terms. Let $(f\alpha_1 \ldots \alpha_m, M)$ and $(f\beta_1 \ldots \beta_m, N)$ be the projections of s and t. Both these marked terms are top-stable and correctly marked and hence, by Lemma 5, they are in S_∞. By condition (c3), they are first-step joinable, i.e. there exist $(f\alpha_1 \ldots \alpha_m, M) \rightarrow^r_{(l_1 \rightarrow r_1)\sigma} (fs'_1 \ldots s'_m, M')$ and $(f\beta_1 \ldots \beta_m, N) \rightarrow^r_{(l_2 \rightarrow r_2)\theta} ft'_1 \ldots t'_m, N')$ such that every s'_i is equivalent to its corresponding t'_i, and $((l_1 \rightarrow r_1), (l_2 \rightarrow r_2))$ is not useless.

We apply these rewrite steps to the original terms to get $fs_1 \ldots s_m \rightarrow^r_{(l_1 \rightarrow r_1)\sigma'} fs''_1 \ldots s''_m = s''$ and $ft_1 \ldots t_m \rightarrow^r_{(l_2 \rightarrow r_2)\theta'} ft''_1 \ldots t''_m = t''$, by choosing σ' and θ' such that for each $i \in \{1 \ldots m\}$, it is the case that (a) if $r_1|_i$ ($r_2|_i$) is a variable not appearing in l_1 (l_2), then $s''_i = t''_i$, and (b) if not, then either s''_i (t''_i) is a constant or it coincides with one of the s_j (t_j) or it coincides with s (t). But it cannot happen that both s''_i and t''_i coincide with s and t, respectively. This is because the rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ are not useless.

By construction, every s''_i is equivalent to its corresponding t''_i. Since s and t are not joinable, s'' and t'' are not joinable, and hence, for some $i \in \{1 \ldots m\}$ we have that s''_i and t''_i are not joinable. This can only happen for case (b) above, and by the previous observation, (s''_i, t''_i) is a smaller witness to non-confluence, a contradiction.

Finally, we are ready to state the main result.

Theorem 1. *Confluence of linear shallow term rewrite systems can be decided in time polynomial in the size of the rewrite system, assuming the maximum arity of any function symbol is bounded by a constant.*

Proof. The input linear shallow rewrite system is transformed into a flat linear rewrite system and then it is saturated under the ordered chaining inference rules. Flattening increases the size $|\mathcal{F}|$ of the signature by a linear factor of the input size. Now, the number of flat linear rewrite rules is bounded by a polynomial in the size $|\mathcal{F}|$ of the signature, and hence these two transformation steps run in polynomial time. Next, the sets S_∞ and J_∞ are computed, again using polynomial time fixpoint computations. Finally, confluence is tested using the characterization given in Lemma 7. The three conditions in Lemma 7 can be tested in polynomial time: (a) Equivalent depth zero terms can be identified because equivalence testing for flat linear rewrite systems can be efficiently done, say using standard completion modulo permutation rules. Joinability of depth zero terms can be tested in polynomial time using simple fixpoint computations, similar to previous work [13]. (b) It is also clear that the conditions (c2) and (c3) can be tested in polynomial time.

Example 3. For the rewrite system R of Example 1, the set S_∞ contains the terms $x + y, 0 + x, x + 0, 0 + 0$, where the positions of 0 are marked. But the pairs $(0, 0 + 0), (x, 0 + x), (x, x + 0)$ are easily seen to be structurally joinable, while $(x + 0, 0 + x)$ is first-step joinable. Hence, this rewrite system is confluent.

3 Relaxing the Restrictions

The reachability, 2-joinability, and confluence problems for shallow term rewrite systems are not known to be decidable. But, we can establish the following lower-bounds.

Theorem 2. *The reachability problem for shallow term rewrite systems is EXPTIME-hard, even when the maximum arity is a constant.*

Proof. We reduce the problem of deciding non-emptiness of language intersection of n bottom-up tree-automata to this problem. The proof is similar to the proof of EXPTIME-hardness of rigid-reachability of shallow terms over ground systems [5]. Let R_1 be the union of the *reversed* transitions of all the n tree-automata. We assume that the tree-automata have disjoint states with accepting states $q_1, q_2, \ldots, q_n$, respectively. Let $R_2 = \{a \to g(q_1, f(q_2, f(q_3, \cdots, f(q_{n-1}, q_n) \cdots))), fxx \to x, gxx \to b\}$, where g, f are two new binary function symbols and a, b are two new constants. Now, a rewrites to b via $R_1 \cup R_2$ iff the intersection of languages accepted by the n automata is nonempty.

For shallow term rewrite systems, EXPTIME-hardness of 2-joinability follows from the hardness of reachability using the reduction in [15]. We next show hardness of deciding confluence of shallow term rewrite systems by modifying the proof of Theorem 2.

Theorem 3. *Deciding confluence of shallow term rewrite systems is EXPTIME-hard, even when the maximum arity is a constant.*

Proof. We add additional rewrite rules to the rewrite system generated in the proof of Theorem 2 to make the system confluent exactly when b is reachable from a. First, we introduce a new constant c in the signature $\mathcal{G}$ of the tree automata and convert all constants d to unary terms $d(c)$. The rules in R_1 are modified to reflect this change. We assume that some ground term can be reached from any tree-automata state q via R_1. Let $R_3 = \{c \to a,\ \ h(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n) \to b \text{ for all } h \in \mathcal{G},\ \ fxb \to b,\ \ fbx \to b,\ \ gxb \to b,\ \ gbx \to b\}$. Now, consider the shallow term rewrite system $R = R_1 \cup R_2 \cup R_3 \cup \{d \to a, d \to b\}$, where R_1 and R_2 are as in proof of Theorem 2 and d is a new constant in the signature. We claim without proof that R is confluent iff the n tree-automata have a non-empty language intersection.

We also note here that reachability, 2-joinability, and confluence problems are undecidable for linear (non-shallow) term rewrite systems [15].

4 Conclusion

In this paper we presented a polynomial time algorithm for deciding confluence of linear shallow term rewrite systems where each variable is allowed at most two occurrences in a rule—one on each side. The time complexity analysis assumes that the maximum arity of a function symbol in the signature is a constant. Our result generalizes those in [2, 13]. We also show that the reachability, joinability and confluence problems are all EXPTIME-hard for shallow non-linear systems, and all three are known to be undecidable for linear non-shallow systems, which indicates that our assumptions can not be easily relaxed without considerably losing efficiency. Our technique can be adapted to decide ground confluence of linear shallow term rewrite systems in polynomial time. It is not clear whether our method can give polynomial time algorithms to decide confluence when we have non-fixed arity or for rule-linear rewrite systems (no variable appears twice in the whole rule) and not necessarily shallow, and this is a matter for future work.

Acknowledgments. We would like to thank the reviewers for their helpful comments.

References

1. L. Bachmair. *Canonical Equational Proofs*. Birkhäuser, Boston, 1991.
2. H. Comon, G. Godoy, and R. Nieuwenhuis. The confluence of ground term rewrite systems is decidable in polynomial time. In *42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, Las Vegas, Nevada, USA, 2001.
3. H. Comon, M. Haberstrau, and J.-P. Jouannaud. Syntacticness, cycle-syntacticness, and shallow theories. *Information and Computation*, 111(1):154–191, 1994.
4. M. Dauchet, T. Heuillard, P. Lescanne, and S. Tison. Decidability of the confluence of finite ground term rewrite systems and of other related term rewrite systems. *Information and Computation*, 88(2):187–201, October 1990.
5. H. Ganzinger, F. Jacquemard, and M. Veanes. Rigid reachability: The non-symmetric form of rigid E-unification. *Intl. Journal of Foundations of Computer Science*, 11(1):3–27, 2000.
6. Guillem Godoy, Robert Nieuwenhuis, and Ashish Tiwari. Classes of Term Rewrite Systems with Polynomial Confluence Problems. *ACM Transactions on Computational Logic (TOCL)*, 2002. To appear.
7. A. Hayrapetyan and R.M. Verma. On the complexity of confluence for ground rewrite systems. In *Bar-Ilan International Symposium On The Foundations Of Artificial Intelligence*, 2001. Proceedings on the web at `http://www.math.tau.ac.il/~nachumd/bisfai-pgm.html`.
8. D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.
9. A. Levy and J. Agusti. Bi-rewriting, a term rewriting technique for monotone order relations. In C. Kirchner, editor, *Rewriting Techniques and Applications RTA-93*, pages 17–31, 1993. LNCS 690.
10. R. Nieuwenhuis. Basic paramodulation and decidable theories. In *11th IEEE Symposium on Logic in Computer Science, LICS 1996*, pages 473–482. IEEE Computer Society, 1996.
11. M. Oyamaguchi. The Church-Rosser property for ground term-rewriting systems is decidable. *Theoretical Computer Science*, 49(1):43–79, 1987.
12. A. Tiwari. Rewrite closure for ground and cancellative AC theories. In R. Hariharan and V. Vinay, editors, *Conference on Foundations of Software Technology and Theoretical Computer Science, FST&TCS '2001*, pages 334–346. Springer-Verlag, 2001. LNCS 2245.
13. A. Tiwari. Deciding confluence of certain term rewriting systems in polynomial time. In Gordon Plotkin, editor, *IEEE Symposium on Logic in Computer Science, LICS 2002*, pages 447–456. IEEE Society, 2002.
14. A. Tiwari. *On the combination of equational and rewrite theories induced by certain term rewrite systems*. Menlo Park, CA 94025, 2002. Available at: `www.csl.sri.com/~tiwari/combinationER.ps`.
15. R. Verma, M. Rusinowitch, and D. Lugiez. Algorithms and reductions for rewriting problems. *Fundamenta Informaticae*, 43(3):257–276, 2001. Also in Proc. of Int'l Conf. on Rewriting Techniques and Applications 1998.
16. H. Zantema. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation*, 17:23–50, 1994.

SRI International

CSL Technical Report SRI-CSL-01-02 (Rev. 2) ● August, 2003

The SAL Language Manual

Leonardo de Moura
Sam Owre
N. Shankar

This report was developed and is maintained by SRI International. SRI's part of the SAL project is funded by DARPA/AFRL contract numbers F30602-96-C-0204 and F33615-00-C-3043.

Computer Science Laboratory ● 333 Ravenswood Ave. ● Menlo Park, CA 94025 ● (650) 326-6200 ● Facsimile: (650) 859-2844

Abstract

SAL stands for Symbolic Analysis Laboratory. It is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. A key part of the SAL framework is a language for describing transition systems. This language serves as a specification language and as the *target* for translators that extract the transition system description for popular programming languages such as Esterel, Java, and Statecharts. The language also serves as a common *source* for driving different analysis tools through translators from the SAL language to the input format for the tools, and from the output of these tools back to the SAL language.

The SAL language was originally designed in collaboration with David Dill of Stanford University and Thomas Henzinger of the University of California at Berkeley. The version presented here is the one currently accepted by the tools developed at SRI.

Contents

Chapter 1

Introduction

SAL stands for Symbolic Analysis Laboratory. It is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. A key part of the SAL framework is a language for describing transition systems. This language serves as a specification language and as the *target* for translators that extract the transition system description for popular programming languages such as Esterel, Java, and Statecharts. The language also serves as a common *source* for driving different analysis tools through translators from the SAL language to the input format for the tools, and from the output of these tools back to the SAL language.

The basic high-level requirements on the SAL language are

1. **Generality:** It should be possible to effectively capture the transition semantics of a wide variety of source languages.

2. **Minimality:** The language should not have redundant or extraneous features that add complexity to the analysis. The language must capture transition system behavior without any complicated control structures.

3. **Semantic Regularity:** The semantics of the language ought to be standard and straightforward so that it is easy to verify the correctness of the various translations with respect to linear and branching time semantics. The semantics should be definable in a formal logic such as PVS.

4. **Language Modularity:** The language should be parametric with respect to orthogonal features such as the type/expression sublanguage, the transition sublanguage, and the module sublanguage.

5. **Compositionality:** The language must have a way of defining transition system modules that can be composed in a meaningful way. Properties of systems composed from modules can then be derived from the individual module properties.

    - **Synchronous composition:** In this form of composition, modules react to inputs synchronously or in zero time, as with combinational circuitry in hardware. In order to achieve semantic hygiene, causal loops arising in such synchronous interactions have to be eliminated. The constraints on the language for the elimination of causal loops should not be so onerous as to rule out sensible specifications.

- **Asynchronous composition:** Modules that are driven by independent clocks are modeled by means of interleaving the atomic transitions of the individual modules.

We present the SAL language in stages consisting of the type system, the expression language, the transition language, modules, synchronous and asynchronous composition of modules, and the specification of systems. The language is largely modular in these choices in the sense that many of the language choices can be independently modified without affecting the other choices. The language is presented in terms of its concrete or presentation syntax but only the internal or abstract syntax is really important for tool interaction.

The SAL language is not that different from the input languages used by various other verification tools such as SMV [3], Murphi [4], Mocha [1], and SPIN [2]. Like these languages, SAL describes transition systems in terms of initialization and transition commands. These can be given by variable-wise definitions in the style of SMV or as guarded commands in the style of Murphi.

Chapter 2

A Simple Example: An N-bit Adder

An N-bit ripple-carry adder module is specified from a one-bit adder module by composing a base one-bit adder module with the synchronous multicomposition of $N-1$ one-bit adder modules. The one-bit adder takes three inputs: the two input bits `a` and `b` and the carry-in bit `cin`, and returns two outputs: the sum bit `sum` and the carry-out bit `cout`. See Figure 2.1. The N-bit adder takes three inputs: the two input bit-vectors `A` and `B` and the carry-in bit `carryin`, and returns two outputs: the sum vector `S` and the carry-out vector `C`. See Figure 2.2.

The `adder` module is definitional, as is usual for a purely combinational circuit description. This means there are no guarded commands, and the adders are synchronously composed.

Note that the requirement that types be nonempty means that the N-bit adder cannot be used to model a 1-bit adder. We plan on allowing empty types in the future, see Section 8.4.

```
adder: CONTEXT =
 BEGIN
  onebitadder: MODULE =
   BEGIN
    INPUT cin, a, b: BOOLEAN
    OUTPUT cout, sum: BOOLEAN
    DEFINITION
      sum = (a XOR b) XOR cin ;
      cout = (a AND b) OR (a AND cin) OR (b AND cin)
   END;

  Nbitadder [N : {n: NATURAL | n > 1}] : MODULE =
   WITH INPUT  A, B : ARRAY [0 .. N-1] OF BOOLEAN, carryin: BOOLEAN;
        OUTPUT S, C : ARRAY [0 .. N-1] OF BOOLEAN
    RENAME a TO A[0], b TO B[0], cin TO carryin,
           sum TO S[0], cout TO C[0] IN
      onebitadder
       ||
       (|| (i : [1 .. N-1]):
          (RENAME a TO A[i], b TO B[i], cin TO C[i-1],
                  sum TO S[i], cout TO C[i] IN
           onebitadder));
END
```

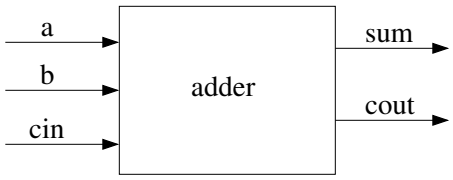


Figure 2.1: Module `adder`

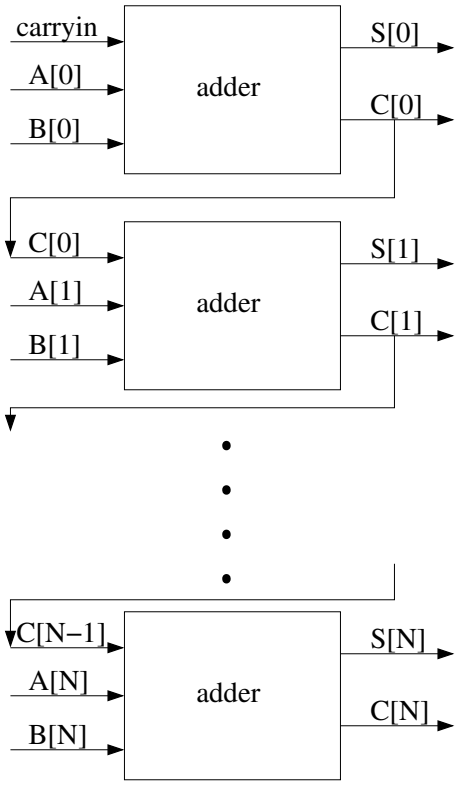


Figure 2.2: Module `Nbitadder`

Chapter 3

The Expression Language

The conventions used in presenting the SAL grammar are that tokens are given in `teletype` font, [*optional*] indicates that *optional* is optional, $\{category\}^{+}_{,}$ indicates one or more occurrences of the syntactic category *category* separated by commas, and $\{category\}^{*}_{,}$ indicates zero or more repetitions of *category* separated by commas. Separators other than comma can be used so that a transition given by a set of named guarded commands separated by the choice operator `[]` can be written as $\{NamedCommands\}^{+}_{[]}$. Nonterminals are written in *italics*.

The SAL language needs to be liberal in order to accommodate translations from other source languages. For this reason, identifiers include a large number of operators. The *special symbols* are parentheses (`(`, `)`), brackets (`[`, `]`), braces (`{`, `}`), the percent sign (`%`), comma (`,`), period (`.`), colon (`:`), semi-colon (`;`), single quote (`'`), exclamation point (`!`), hash (`#`), question mark (`?`), and underscore (`_`). Tokens can be separated by *WhiteSpace*, which consists of spaces, tabs, carriage returns, and line feeds.

SpecialSymbol	:=	`(` | `)` | `[` | `]` | `{` | `}` | `%` | `,` | `.` | `;` | `:` | `'` | `!` | `#` | `?` | `_`
Letter	:=	`a` | ... | `z` | `A` | ... | `Z`
Digit	:=	`0` | ... | `9`
Identifier	:=	*Letter* {*Letter* | *Digit* | `?` | `_`}*
		| {*Opchar*}$^{+}$
Numeral	:=	{*Digit*}$^{+}$

An *Opchar* is any character that is not a *Letter*, *Digit*, *SpecialSymbol*, or *WhiteSpace*. For example, `f1_3` and `+-+` are identifiers, but `a+-1` is three tokens: two identifiers (`a` and `+-`), and a numeral.

The grammar is case-sensitive. The reserved words must be in upper case. The reserved words are:

`AND`, `ARRAY`, `BEGIN`, `BOOLEAN`, `CLAIM`, `CONTEXT`, `DATATYPE`, `DEFINITION`, `ELSE`, `ELSIF`, `END`, `ENDIF`, `EXISTS`, `FALSE`, `FORALL`, `GLOBAL`, `IF`, `IN`, `INITIALIZATION`, `INPUT`, `INTEGER`, `LAMBDA`, `LEMMA`, `LET`, `LOCAL`, `MODULE`, `NATURAL`, `NOT`, `NZINTEGER`, `NZREAL`, `OBLIGATION`, `OF`, `OR`, `OUTPUT`, `REAL`, `RENAME`, `THEN`, `THEOREM`, `TO`, `TRANSITION`, `TRUE`, `TYPE`, `WITH`, `XOR`.

Comments in SAL are preceded by the `%` symbol and terminated by an end-of-line.

3.1 Types[1]

The SAL language supports the built-in basic types for booleans, natural numbers, integers, and reals. New basic types may be introduced using uninterpreted type declarations. Types may be used in type constructions to create subtype, subrange, array, function, tuple, and record types. Function, tuple, and record types may be dependent. In addition to uninterpreted type declarations, that introduce a name without a defining form, type declarations may be used to introduce names for existing types, as well as scalars and datatypes. The grammar for types is given by

TypeDef	:=	*Type*
		| *ScalarType*
		| *DataType*
Type	:=	*BasicType*
		| *Name*
		| *Subrange*
		| *SubType*
		| *ArrayType*
		| *TupleType*
		| *FunctionType*
		| *RecordType*
		| *StateType*
BasicType	:=	`BOOLEAN` | `REAL` | `INTEGER` | `NZINTEGER` | `NATURAL` | `NZREAL`
Name	:=	*Identifier*
QualifiedName	:=	*Identifier*[{*ActualParameters*}]`!` *Identifier*
Subrange	:=	`[` *Bound* `..` *Bound* `]`
SubType	:=	`{` *Identifier* `:` *Type* `|` *Expression* `}`
Bound	:=	*Unbounded* | *Expression*
Unbounded	:=	`_`
ArrayType	:=	`ARRAY` *IndexType* `OF` *Type*
IndexType	:=	`INTEGER` | *Subrange* | *ScalarTypeName*
ScalarTypeName	:=	*Name*
TupleType	:=	`[` *VarType* `,` {*VarType*}$^{+}_{,}$ `]`
FunctionType	:=	`[` *VarType* `->` *Type* `]`
VarType	:=	[*Identifier* `:`] *Type*
RecordType	:=	`[#` {*Identifier* `:` *Type*}$^{+}_{,}$ `#]`
StateType	:=	*Module* `.` `STATE`
ScalarType	:=	`{`{*Identifier*}$^{+}_{,}$`}`
DataType	:=	`DATATYPE` *Constructors* `END`
Constructors	:=	{*Identifier*[`(` *Accessors*`)`]}$^{+}_{,}$
Accessors	:=	{*Identifier* `:` *Type*}$^{+}_{,}$

A *TypeDef* is a type expression that can occur as the body of a type declaration, whereas a *Type* is more restrictive and circumscribes the types that can be used within an expression or a transition system module. Two types are equivalent if they are identical modulo the renaming of bound variables, the rearrangement of record labels, the equality of subtype predicates, and the unfolding of the definitions of defined types that are not scalar types or datatypes. Equivalence for types that are defined to be uninterpreted, scalar types, and datatypes is just name equivalence. Name equivalence is not a simple concept because compound names consist of the context name, actual

[1] SAL types are very similar to PVS types, both syntactically and semantically. See the PVS Language Reference [5].

parameters, and the identifier. Two names are equivalent if they agree on the context name, and the identifier, and the actual parameters, which are either types or expressions, are equivalent. Types in SAL (as in PVS) are modeled as sets, and two types are equivalent when every element of one is an element of the other. Thus the dependent types

```
[# a: INTEGER, b: {x: INTEGER | x < a} #]
[# b: INTEGER, a: {x: INTEGER | b < x} #]
```

are equivalent, and similarly for tuples. One way to see this equivalence is to note that each is equivalent to the type

```
{r: [# a: INTEGER, b: INTEGER #] | r`b < r`a}
```

Note that in an array type, the index type must either be `INTEGER`, a subrange, or a scalar type. SAL has a higher-order type system since it contains function types between arbitrary domain and range types. SAL types need not be finite, and the `REAL` and `INTEGER` types, for example, are infinite. The `REAL` type is the mathematical reals, not a floating point representation. Arrays with infinite index and range types are also admissible.

There are a fixed set of subtyping relations among the types that naturally corresponds to a subset relation between the denotations of these types. The subrange type `[a .. b]` is an abbreviation for {`x: INTEGER | a <= x AND x <= b`}, `[a .. _]` is an abbreviation for {`x: INTEGER | a <= x`}, and `[_ .. b]` is an abbreviation for {`x: INTEGER | x <= b`}. The type `NATURAL` is merely an abbreviation for {`x: INTEGER | 0 <= x`}. Any subrange is a subtype of a larger subrange. It is also a subtype of `INTEGER`. An array (function) type A is a subtype of another array (function) type B if the index types are identical, and the range type of A is a subtype of the range type of B. Similarly, a record type A is a subtype of another record type B if every element of A is an element of B, which means the label sets must be the same, though as described in type equivalence, the corresponding types do not have to be in the subtype relation.

A *StateType* is a record type representing the state of the specified module. This is described in more detail below.

All types must be checked to be nonempty through the possible generation of proof obligations entailing nonemptiness.

Recursive datatypes can be used to define list and tree-like types. The datatype is specified by a list of constructor operations, each with a list of accessor operations. For example, the list type of integers is constructed as

```
intlist: TYPE = DATATYPE
                  cons(car : INTEGER, cdr : intlist),
                  nil
                END
```

Recognizers are automatically generated by appending a `?` to the corresponding constructor. Thus `cons?` and `nil?` are recognizers for `intlist`. These may be used in definitions. For example, `length` may be defined recursively[2] as

[2]This will lead to proof obligations showing that the function is total, i.e., terminating.

```
length: [intlist -> NATURAL] =
   LAMBDA (lst: intlist):
     IF nil?(lst) THEN 0 ELSE 1 + length(cdr(lst)) ENDIF
```

3.2 Expressions

Expressions in the SAL language have to be type-correct with respect to the types in the type language. The expressions consist of constants, variables, applications with Boolean, arithmetic, and bit-vector operations, and array, function, tuple, and record selection and updates. Conditional (if-then-else) expressions are also part of the expression language.

Expression := *NameExpr*
| *QualifiedNameExpr*
| *NextVariable*
| *Numeral*
| *Application*
| *InfixApplication*
| *ArraySelection*
| *RecordSelection*
| *TupleSelection*
| *UpdateExpression*
| *LambdaAbstraction*
| *QuantifiedExpression*
| *LetExpression*
| *SetExpression*
| *ArrayLiteral*
| *RecordLiteral*
| *TupleLiteral*
| *Conditional*
| (*Expression*)
| *StatePred*

NameExpr := *Name*
QualifiedNameExpr := *QualifiedName*
NextVariable := *Identifier* '
Application := *Function Argument*
Function := *Expression*
Argument := ({ *Expression* }$^{+}_{,}$)
InfixApplication := *Expression Identifier Expression*
ArraySelection := *Expression* [*Expression*]
RecordSelection := *Expression* . *Identifier*
TupleSelection := *Expression* . *Numeral*
UpdateExpression := *Expression* WITH *Update*
Update := *UpdatePosition* := *Expression*
UpdatePosition := { *Argument* | [*Expression*] | . *Identifier* | . *Numeral* }$^{+}$
LambdaAbstraction := LAMBDA (*VarDecls*) : *Expression*
VarDecls := { *VarDecl* }$^{+}_{;}$
VarDecl := { *Identifier* }$^{+}_{,}$: *Type*
QuantifiedExpression := *Quantifier* (*VarDecls*) : *Expression*
Quantifier := FORALL | EXISTS
LetExpression := LET *LetDeclarations* IN *Expression*
LetDeclarations := { *Identifier* : *Type* = *Expression* }$^{+}_{;}$
SetExpression := *SetListExpression* | *SetPredExpression*
SetPredExpression := { *Identifier* : *Type* | *Expression* }
SetListExpression | { { *Expression* }$^{+}_{,}$ }
ArrayLiteral := [[*IndexVarDecl*] *Expression*]
IndexVarDecl := *Identifier* : *IndexType*
RecordLiteral := (# { *RecordEntry* }$^{+}_{,}$ #)
RecordEntry := *Identifier* := *Expression*
TupleLiteral := *Argument*
Conditional := IF *Expression ThenRest*
ThenRest := THEN *Expression*
[*ElsIf*]
ELSE *Expression* ENDIF
ElsIf := ELSIF *Expression ThenRest*
StatePred := *Module* . (INIT | TRANS)

The unary operators include boolean negation NOT, and integer minus -.

The binary operators include

- Polymorphic equality = and disequality /=. Note that since subtypes are semantically the same as subsets, equality and disequality are defined on the maximal supertype of a type.
- Boolean operations of conjunction AND, disjunction OR, implication =>, equivalence <=>, and exclusive-or XOR
- Real arithmetic operations of addition +, subtraction -, multiplication *, division /, and the comparison operators <, <=, >, >=. Note that the divisor type of division is restricted to NZREAL and the type rules generate a proof obligation if the divisor is not known to be nonzero. The integer arithmetic operations of DIV and MOD are included in the binary operations. Both require nonzero integers, i.e., NZINTEGER, in the divisor position and they satisfy the equation

$$a = b * (a \texttt{ DIV } b) + (a \texttt{ MOD } b)$$

Although the parser allows any *Identifier* as an infix operator, it is clearly useful to have a standard operator precedence so that expressions such as `y + 1 = x AND A` are not parsed nonsensically, e.g., as `y + (1 = (x AND A))`. The precedence is as follows, from lowest to highest:

`<=>`
`=>`
`OR`, `XOR`
`AND`
`=`, `/=`
`>`, `>=`, `<`, `<=`
OtherIdentifier
`+`, `-`
`*`, `/`

`<=>`, `OR`, `XOR`, `AND`, `+`, infix `-`, `*`, and `/` are all left-associative, `=>` is right-associative, and the rest are non-associative.

The *LetExpression* is parallel, to get the sequential form use nested `LET`s, e.g.,

```
LET a = f(b) IN
 LET b = f(a) IN e
```

The proof obligations generated during typechecking are called type correctness conditions (TCCs). In addition to operations with subtype domains such as division, the sources of TCCs include expressions of subrange types, recursive datatypes, recursive definitions, and type nonemptiness.

An expression without *NextVariable*s is called a *current expression* and is represented by the nonterminal *CExpression*. We will not define its grammar but it essentially corresponds to the grammar for *Expression* with the occurrences of *NextVariable* removed.

SAL expressions contain two kinds of variables: logical variables and state variables. The state variables are either current variables or *NextVariables*. SAL types and expressions are given a semantics with respect to a model $\mathcal{M}$ that fixes the meanings of types, constants, and operators, an assignment ρ of values to the free logical variables, and an assignment of values to the current variables x and the *NextVariable*s x' by a pair of *states* $\langle r, s\rangle$. The meaning of expression e with respect to model $\mathcal{M}$, assignment ρ, and a pair of states $\langle r, s\rangle$, is given by $\mathcal{M}[\![e]\!]^{\rho}_{\langle r,s\rangle}$. If variable x has type A, then the interpretation of x in state s, $s(x)$, must be an element of $\mathcal{M}[\![A]\!]$. If x is a variable in the state type, then $\mathcal{M}[\![x]\!]_{\langle r,s\rangle} = r(x)$, and $\mathcal{M}[\![x']\!]_{\langle r,s\rangle} = s(x)$. The interpretation of types and operators are the standard ones. When expression e does not contain any *NextVariable*s, we write the meaning of e as $\mathcal{M}[\![e]\!]_r$.

The *StatePred* expressions provide access to the initialization predicate and transition relations for a given module M. In particular, M`.INIT` is of type [M`.STATE -> BOOLEAN`] and M`.TRANS` is of type [M`.STATE,` M`.STATE -> BOOLEAN`].

Chapter 4

The Transition Language

A transition system *module* consists of a *state* type, an *invariant definition* on this state type, an *initialization condition* on this state type, and a binary *transition relation* of a specific form on the state type. The state type is defined by four pairwise disjoint sets of *input*, *output*, *global*, and *local* variables. The input and global variables are the *observed* variables of a module and the output, global, and local variables are the *controlled* variables of the module. The language constructs for defining modules from transition systems are treated in Chapter 5.

The transition rules are constraints on the current and next states of the transition. The current variables are written as `X` whereas the next state variables are written as `X'`.

4.1 Definitions

Definitions are the basic constructs used to build up the invariants, initializations, and transitions of a module. Definitions are used to specify the trajectory of variables in a computation by providing constraints on the controlled variables in a transition system. For variables ranging over aggregate data structures like records or arrays, it is possible to define each component separately. For example,

```
x' = x + 1
```

simply increments the state variable `x`, where `x'` is the newstate of the variable,

```
y'[i] = 3
```

sets the new state of the array `y` to be `3` at index `i`, and to remain unchanged on all other indices, and

```
z.foo.1[0] = y
```

constrains state variable `z`, which is a record whose `foo` component is a tuple, whose first component in turn is an array of the same type as `y`.

The left-hand side of a definition is given by the nonterminal *Lhs*.

Lhs := *Identifier* ['] {*Access*}*
Access := *ArrayAccess* | *RecordAccess* | *TupleAccess*
ArrayAccess := [*Expression*]
RecordAccess := . *Identifier*
TupleAccess := . *Numeral*

Simple definitions are of the form

SimpleDefinition := *Lhs RhsDefinition*
RhsDefinition := *RhsExpression* | *RhsSelection*
RhsExpression := = *Expression*
RhsSelection := IN *Expression*

For an *RhsExpression*, the *Lhs* is simply assigned the corresponding value. For an *RhsSelection*, the *Lhs* is assigned any value satisfying the expression, which must be a predicate (a boolean-valued *LambdaAbstraction* or a *SetExpression*). This predicate must be satisfiable; an *invariant obligation* is generated if it cannot be determined to be nonempty.

Note that in an *Access*, all unspecified components are unchanged, thus `x'[i].name = Ed` is equivalent to `x' = x WITH [i].name := Ed`. If the given transition has multiple assignments to `x`, they must all be collected to get the equivalent form, for example, the assignments

```
x'[0].name = Ed;
x'[1].name = Al
```

are equivalent to `x' = x WITH [0].name = Ed WITH [1].name = Al`.

There are other restrictions on the *Access*. Within a given `DEFINITION`, `INITIALIZATION`, or `TRANSITION` section of a module the *Lhs* accesses must all be unique. Thus the assignments

```
x'[3] = 0;
x'[f(3)] = 0
```

will generate a proof obligation that `3 /= f(3)`. Note that it does not matter that these are really the same assignments if they are equal, the obligation will still be generated.

A transition equation in the `TRANSITION` section defines a *NextVariable* on the left-hand side in terms of an expression that can contain *NextVariable* occurrences. A *SimpleDefinition* can occur in the `TRANSITION` section of a transition system. An array index expression on the left-hand side must not contain any state variables.

Definitions := {*Definition*}$^{+}_{;}$
Definition := *SimpleDefinition* | *ForallDefinition*
ForallDefinition := (FORALL (*VarDecls*): *Definitions*)

In a transition system module, a controlled variable must be defined exactly once. It is easy to write definitions that admit causal cycles such as:

```
X = NOT Y;
Y = X
```

Such causal loops can lead to contradictory or meaningless definitions and have to be ruled out. One way to avoid causal loops is by means of an ordering on the variables so that the right-hand side of a definition can contain only those variables that are lower in the ordering. However, such a restriction would rule out natural definitions where variables can depend on each other without triggering a causal loop, for example

```
X = IF A THEN NOT Y ELSE C ENDIF
Y = IF A THEN B ELSE X ENDIF
```

Here there is no causal loop since `X` depends on `Y` only when `A` holds, and `Y` depends on `X` only when `NOT A` holds. A dependency analysis generates a Boolean formula indicating the governing conditions *GC(X, Y)* under which a variable `X` immediately depends on another variable `Y`. The governing conditions are required to be current expressions. For example, $GC(\mathtt{X},\mathtt{Y})$ for the above definitions of `X` yields `A`. If there is no assignment defining `X` in terms of `Y` then *GC(X, Y)* is *false*. Then $GC^*(\mathtt{X},\mathtt{Y})$ yields the governing conditions under which a variable `X` could indirectly depend on a variable `Y`. For example, if `X` depends on a variable `Z` that in turn depends on `Y`, then $GC^*(\mathtt{X},\mathtt{Y})$ is just $GC(\mathtt{X},\mathtt{Y}) \vee (GC(\mathtt{X},\mathtt{Z}) \wedge GC(\mathtt{Z},\mathtt{Y}))$. Thus, in the above definitions of `X` and `Y`, $GC^*(X, X)$ is $\mathtt{A} \wedge \neg\mathtt{A}$. The dependency conditions can be used to generate the conditions $C_{\mathtt{X}}$ under which a variable `X` could depend on itself. For such dependency loops to be avoided, the condition $C_{\mathtt{X}}$ must be shown to be invariantly false in the transition system. In the above example, $C_{\mathtt{X}}$ would be the obviously unreachable assertion $\mathtt{A} \wedge \neg\mathtt{A}$. The dependency analysis (causality checks) generate proof obligations to this effect. A similar dependency analysis can be carried out for initialization definitions and transition definitions.

4.2 Guarded Commands

Definitions are convenient for specifying the values taken on by those controlled variables whose transitions can be independently specified in a simple equational form. Definitions have some drawbacks. For variables whose definitions follow a similar case structure, this case structure has to be repeated in each of the definitions. For such controlled variables, it is convenient to specify their initialization and transitions in terms of guarded commands. Each guarded command consists of a *guard* formula and an assignment part. The guard is a boolean expression in the current controlled (local, global, and output) variables and current and next state input variables. The assignment part is a list of equalities between a left-hand side next state variable and a right-hand side expression in current and next state variables.

$$\begin{array}{rcl} \mathit{GuardedCommand} & := & \mathit{Guard}\ \texttt{-->}\ \mathit{Assignments} \\ \mathit{Guard} & := & \mathit{Expression} \\ \mathit{Assignments} & := & \{\mathit{SimpleDefinition}\}^{*}_{;}\ [\ ;\ ] \end{array}$$

Note that both the initializations and transitions may be specified by guarded assignments. No variable that is defined in the *Lhs* of a definition can be assigned in either a guarded initialization or transition. The initializations must not contain next state variables, whereas the transitions must have next state variables on the left-hand side of assignments, and may have next state variables on the right-hand side. The well-formedness checks on the guarded transitions are that the guard must not contain controlled next state variables, i.e., `X'` for some controlled variable `X`, since these

variables are only assigned values in the assignment part. The assignments in the assignment part must ensure that no controlled variable is assigned more than once.

The causality checks and proof obligations corresponding to a guarded initialization or transition are similar to those for definitions. The primary difference is that current conjuncts in the guard can be conjoined to the the conditions when the proof obligations are generated. For example, if there is a guarded command of the form g `-->` *Assignments* where the dependency analysis on the combination of the *Assignments* and the definitions yields the conditions for a causal loop on variable `X` as C_X, then the conjunction $g \wedge C_X$ must be shown to be unreachable.

Note that the initialization and transition sections may contain simple definitions and/or guarded commands. The model of execution is that when the module gets activated, one guarded transition is chosen so that the guard formula holds in the current (and possibly next input) state, and the transition is the conjunction of the associated guarded transition with all the definitions of the transition section(s). If no guard is satisfied, the module may deadlock. A synchronously composed system is deadlocked if any of its component modules is. An asynchronously composed system is only deadlocked if all its components are. If you want to ensure a given module does not deadlock, just make sure that there is always some guard of the module that hlods true (the `ELSE` clause is useful for this).

Chapter 5

The Module Language

A module is a self-contained specification of a transition system in SAL. Modules can be independently analyzed for properties and composed synchronously or asynchronously. Here is a fairly simple module declaration.

```
m : MODULE =
 BEGIN
  INPUT temp: INTEGER
  LOCAL high: BOOLEAN, ctr: NATURAL
  OUTPUT danger: BOOLEAN
  DEFINITION high = i > 100
  INITIALIZATION ctr = 0; danger = FALSE
  TRANSITION [  ctr > 3 --> danger' = danger OR high
             [] ctr <= 3 AND high --> ctr' = ctr + 1
             [] ELSE --> ctr' = 0
              ]
 END
```

Here `m` is a *BaseModule*, that is intended to monitor the temperature and indicate a problem if the temperature stays high for too long. It declares the input variable `temp`, local variables `high` and `ctr`, and output variable `danger`. Initially `danger` is `FALSE` and `ctr` is 0, and when this module is activated it sets `danger` to `TRUE` if `temp` exceeds 100 more than 3 times in a row.

Once base modules are declared, they may be composed synchronously or asynchronously to yield new modules. The grammar for module expressions is given below. The grammars for *Definitions* and *GuardedCommand* are described in the previous chapter, but are repeated here for convenience.

```
Module ::= BaseModule
         | ModuleInstance
         | SynchronousComposition
         | AsynchronousComposition
         | MultiSynchronous
         | MultiAsynchronous
         | Hiding
         | NewOutput
         | Renaming
         | WithModule
         | ObserveModule
         | ( Module )

BaseModule ::= BEGIN BaseDeclarations END
BaseDeclarations ::= {BaseDeclaration}*
BaseDeclaration ::= InputDecl
                  | OutputDecl
                  | GlobalDecl
                  | LocalDecl
                  | DefDecl
                  | InitDecl
                  | TransDecl
InputDecl ::= INPUT VarDecls
OutputDecl ::= OUTPUT VarDecls
GlobalDecl ::= GLOBAL VarDecls
LocalDecl ::= LOCAL VarDecls
DefDecl ::= DEFINITION Definitions
InitDecl ::= INITIALIZATION {DefinitionOrCommand}+; [ ; ]
TransDecl ::= TRANSITION {DefinitionOrCommand}+; [ ; ]

DefinitionOrCommand ::= Definition
                      | [ SomeCommands ]
Definitions ::= {Definition}+;
Definition ::= SimpleDefinition | ForallDefinition
ForallDefinition ::= (FORALL ( VarDecls): Definitions)
SimpleDefinition ::= Lhs RhsDefinition
Lhs ::= Identifier[ ' ] {Access}*
Access ::= ArrayAccess | RecordAccess | TupleAccess
ArrayAccess ::= [ Expression ]
RecordAccess ::= . Identifier
TupleAccess ::= . Numeral
RhsDefinition ::= RhsExpression | RhsSelection
RhsExpression ::= = Expression
RhsSelection ::= IN Expression
SomeCommands ::= {SomeCommand}+[] [ [] ElseCommand ]
SomeCommand ::= NamedCommand | MultiCommand
NamedCommand ::= [ Identifier : ] GuardedCommand
GuardedCommand ::= Guard --> Assignments
Guard ::= Expression
Assignments ::= {SimpleDefinition}*; [ ; ]
MultiCommand ::= ([] ( VarDecls): SomeCommand)
ElseCommand ::= [ Identifier : ] ELSE --> Assignments
```

$$
\begin{array}{rcl}
\mathit{ModuleInstance} & := & \{\mathit{ModuleName} \mid \mathit{QualifiedModuleName}\}\ \mathit{Name}[\ \texttt{[}\{\mathit{Expression}\}^{+}_{,}\texttt{]}\ ] \\
\mathit{ModuleName} & := & \mathit{Name} \\
\mathit{QualifiedModuleName} & := & \mathit{QualifiedName} \\
\mathit{SynchronousComposition} & := & \mathit{Module}\ \texttt{||}\ \mathit{Module} \\
\mathit{AsynchronousComposition} & := & \mathit{Module}\ \texttt{[]}\ \mathit{Module} \\
\mathit{MultiSynchronous} & := & \texttt{(||}\ \mathit{(Identifier : IndexType)}\texttt{:}\ \mathit{Module)} \\
\mathit{MultiAsynchronous} & := & \texttt{([]}\ \mathit{(Identifier : IndexType)}\texttt{:}\ \mathit{Module)} \\
\mathit{Hiding} & := & \texttt{LOCAL}\ \{\mathit{Identifier}\}^{+}_{,}\ \texttt{IN}\ \mathit{Module} \\
\mathit{NewOutput} & := & \texttt{OUTPUT}\ \{\mathit{Identifier}\}^{+}_{,}\ \texttt{IN}\ \mathit{Module} \\
\mathit{Renaming} & := & \texttt{RENAME}\ \mathit{Renames}\ \texttt{IN}\ \mathit{Module} \\
\mathit{Renames} & := & \{\mathit{Lhs}\ \texttt{TO}\ \mathit{Lhs}\}^{+}_{,} \\
\mathit{WithModule} & := & \texttt{WITH}\ \mathit{NewVarDecls}\ \mathit{Module} \\
\mathit{NewVarDecls} & := & \{\mathit{InputDecl} \mid \mathit{OutputDecl} \mid \mathit{GlobalDecl}\}^{+}_{;} \\
\mathit{ObserveModule} & := & \texttt{OBSERVE}\ \mathit{Module}\ \texttt{WITH}\ \mathit{Module}
\end{array}
$$

5.1 Base Modules

A *BaseModule* identifies the pairwise distinct sets of input, output, global, and local variables. This characterizes the *state* of the module.

As described below, base modules also may consist of several sections. Note that the grammar allows variables and sections to be given in any order, and there may, for example, be 3 distinct `TRANSITION` sections. In every case, it is the same as if there was a prescribed order, with each class of variable and section being the union of the individual declarations.

`DEFINITION` section. Definitions appearing in the `DEFINITION` section(s) are treated as invariants for the system. When composed with other modules, the definitions remain true even during the transitions of the other modules. For this reason, proof obligations may be generated for a composition where definition sections are involved. This section is usually used to define controlled variables whose values ultimately depend on the inputs, for example, a boolean variable that becomes true when the temperature goes above a specified value.

Definition sections must be used with care, especially when modeling asynchronous systems, as this means that in some sense the execution of a module on a remote machine can still be seen locally.

`INITIALIZATION` section. The `INITIALIZATION` section(s) constrain the possible initial values for the local, global, and output declarations. Input variables may not be initialized. The `INITIALIZATION` section(s) determine a state predicate that holds of the initial state of the base module.

Definitions and guarded commands appearing in the `INITIALIZATION` section must not contain any *NextVariable* occurrences, i.e., both sides of the defining equation must be *current expressions*. Guards may refer to any variables, this acts as a form of postcondition when controlled variables are involved. This is like backtracking: operationally a guarded initialization is selected, the assignments made, and if the assignments violate the guard the assignments are undone and a new guarded initialization is selected.

`TRANSITION` section. The `TRANSITION` section(s) constrain the possible next states for the local, global, and output declarations. As this is generally defined relative to the previous state of the module, the transition section(s) determine a state relation. Input variables may not appear on the *Lhs* of any assignments. Guards may refer to any variables, even *NextVariable*s. As with guarded initial transitions, guards involving *NextVariable*s have to be evaluated after the assignments have been made, and if they are false the assignments must be undone and a new guarded transition selected.

5.2 State Variable Manipulation

Output and global variables can be made local by the `LOCAL` construct. Global variables can be made output by the `OUTPUT` construct. In order to avoid name clashes, variables in a module can be renamed using the `RENAME` construct. When the renaming variable is an identifier, its type can be easily inferred from the renamed variable. New state variables used for renaming can be introduced using the `WITH` construct for `INPUT`, `OUTPUT`, and `GLOBAL` declarations. These newly declared variables can be used in the `RENAME` construct to rename the variables in a given module. The renaming should be consistent so that the input variables can be renamed only by input variables, output variables only by output variables, and global variables only by output or global variables. The types of the renamed and the renaming variable should also match.

5.3 Module Composition

Modules can be combined by either synchronous or asynchronous composition.

Let module M_i consists of input variables I_i, output variables O_i, global variables G_i, and local variables L_i. The module $M_1 \,||\, M_2$ and $M_1 \,[]\, M_2$ respectively represent the synchronous and asynchronous composition of M_1 and M_2.

Variables with the same identifier are treated as identical, and it is an error to compose modules that assign different types to the same identifier. The syntactic constraints on both synchronous and asynchronous composition are that the output variable sets must be disjoint from the global and output variables of the other module ($O_1 \bigcap (O_2 \bigcup G_2) = \emptyset$, $(O_1 \bigcup G_1) \bigcap O_2 = \emptyset$), the local variables must be disjoint from the other variables ($L \bigcap (I \bigcup O \bigcup G) = \emptyset$), but need not be disjoint from each other.

The input variables I, the output variables O, global variables G, and the local variables L of $M_1 \,||\, M_2$ and $M_1 \,[]\, M_2$ are given by

$$\begin{aligned} I &= (I_1 \bigcup I_2) - (O \bigcup G) \\ O &= (O_1 \bigcup O_2) \\ G &= (G_1 \bigcup G_2) \\ L &= (L_1 \bigcup L_2) \end{aligned}$$

The semantics of synchronous composition is that the module $M_1 \,||\, M_2$ consists of initializations that are the combination of initializations from the two modules, and the transitions are the combinations of the individual transitions of the two modules. The definitions of $M_1 \,||\, M_2$ are simply the union

of the definitions in M_1 and M_2. The initializations of $M_1 \,||\, M_2$ are the pairwise combination of the initializations in M_1 and M_2. Two guarded initializations are combined by conjoining the guards and by taking the union of the assignments. Let $g_{1,i}$ `-->` $a_{1,i}$ be an initialization from M_1 and $g_{2,j}$ `-->` $a_{2,j}$ be an initialization from M_2. The guard $g_{1,i}$ might contain output variables of M_2, and similarly, guard $g_{2,j}$ might contain output variables of M_1. For the combination to be sensible, only at most one of these guards, say $g_{1,i}$, is allowed to contain output variables of the other module. If we take $\overline{a_{2,j}}$ as the union of the assignments in $a_{2,j}$ with the initialization definitions of M_2, then we can repeatedly apply $\overline{a_{2,j}}$ as a substitution. It should then be the case that the repeated application $\overline{a_{2,j}}^*(g_{1,i})$ converges. The combination of the two initializations is then $\overline{a_{2,j}}^*(g_{1,i}) \wedge g_{2,j}$ `-->` $a_{1,i}\,;a_{2,j}$. The resulting combination might not be sensible since the conjunction of the guards could be inconsistent. The combination of the assignments $a_{1,i}\,;a_{2,j}$ might also be causally inconsistent and proof obligations have to be generated to ensure that such combinations do not occur. The dependency analysis in the case of synchronous composition is similar to that for a single module with the restriction that only cycles involving variables from both modules need be considered.

The consistency and dependency analysis for combinations of guarded transitions in a synchronous composition is similar to that for guarded initializations. In this manner, the synchronous composition $M_1 \,||\, M_2$ of two modules M_1 and M_2 can be expressed as a single module combining the definitions, initializations, and transitions from the individual modules. If there are n_1 guarded commands in M_1 and n_2 in M_2, the composition $M_1 \,||\, M_2$ could have up to $n_1 * n_2$ guarded commands. Thus it is not always feasible to expand out the module corresponding to such a composition. The expectation is that this will rarely be necessary since the modules can be individually analyzed and the properties composed.

The semantics of asynchronous composition of two modules is given by the conjunction of the initializations and the interleaving of the transitions of the two modules. For this purpose, the definitions in M_1 and M_2 must first be eliminated by including them in the guarded initializations and transitions. The module corresponding to $M_1 \,[]\, M_2$ is obtained by combining the initializations as in synchronous composition and taking the union of the transition definitions and the guarded transitions. The combination of initializations can generate proof obligations but there are no new proof obligations arising from the union of the module transitions.

The form of composition in SAL supports a compositional analysis in the sense that any module properties expressed in linear-time temporal logic or in the more expressive universal fragment of CTL* are preserved through composition. A similar claim holds for asynchronous composition with respect to stuttering invariant properties where a stuttering step is one where the local and output variables of the module remain unchanged.

The causality analysis for synchronous multicompositions is carried out inductively by unfolding the multicomposition into a composition of a single module and a smaller multicomposition.

5.4 Module Declarations

It is good pragmatics to name a module. This name can be used to index the local variables so that they need not be renamed during composition. Also, the properties of the module can be indexed on the name for quick look-up. Parametric modules allow the use of logical (state-independent) and type parameterization in the definition of modules. A parametric module is defined as

ModuleDeclaration := *Identifier*[[*VarDecls*]] : MODULE = *Module*

Parametric modules allow modules to be defined with some open parameters that can be instantiated when the module is used.

Chapter 6

SAL Contexts

The language so far can describe transition system modules but has no way of declaring new types or constants or asserting properties of these modules. The SAL context language provides the framework for declaring types, constants, modules, and module properties. Below we present the syntax for contexts containing declarations for constants, types, modules, assertions, and other (imported) contexts. SAL contexts are read from left to right, top to bottom, and an entity must be declared before it is referenced.[1]

There is no name overloading in SAL. An unqualified name always refers to the local context. Qualified names must provide both the context and the parameters. Because of this, explicit importings are not needed.[2]

Context	:=	*Identifier* [{*Parameters*}] *:* `CONTEXT` = *ContextBody*
Parameters	:=	[*TypeDecls*] `;` {*VarDecls*}$^*_,$
TypeDecls	:=	{*Identifier*}$^+_,$ `:` `TYPE`
ContextBody	:=	`BEGIN` *Declarations* `END`
Declarations	:=	{*Declaration* `;`}$^+$
Declaration	:=	*ConstantDeclaration*
		| *TypeDeclaration*
		| *AssertionDeclaration*
		| *ContextDeclaration*
		| *ModuleDeclaration*
ConstantDeclaration	:=	*Identifier* [`(` *VarDecls* `)`] `:` *Type* [`=` *Expression*]
TypeDeclaration	:=	*Identifier* `:` `TYPE` [`=` *TypeDef*]
AssertionDeclaration	:=	*Identifier* `:` *AssertionForm* `=` *AssertionExpression*
AssertionForm	:=	`OBLIGATION` | `CLAIM` | `LEMMA` | `THEOREM`
ContextDeclaration	:=	*Identifier* `:` `CONTEXT` `=` *Identifier*{*ActualParameters*}
ActualParameters	:=	{*Type*}$^*_,$ `;` {*Expression*}$^*_,$

[1] For those readers familiar with PVS, a SAL context is very similar to a PVS theory, but with different sets of allowable declarations.

[2] We are considering adding IMPORTINGs for convenience in the concrete language, but the parser should always be able to generate fully qualified names in the abstract syntax. See Section 8.2.1

6.1 Context Parameters

Context parameters allow for generic contexts that may be used from other contexts with different instances. Thus a context may be parameterized by a positive integer `N` that gives the number of processes, and a modelchecker may instantiate this to 6, in order to make it finite.

Within the given context, parameter types are treated as uninterpreted types, and parameter variables are treated as uninterpreted constants. Note that distinct type parameters are treated as distinct types, although they may be instantiated to the same type.

6.2 Constant Declarations

The simplest constant declaration provides an uninterpreted constant, e.g.,

```
c: INTEGER
```

Note that because all types must be nonempty, no proof obligation will be generated for the constant, though there may be one generated for the type.

Constant declarations may also provide a definition:

```
n: INTEGER = 3
f: [INTEGER -> [INTEGER -> INTEGER]] =
   LAMBDA (x: INTEGER): LAMBDA (y: INTEGER): x + n * y
```

A defining form may be used, which is usually more readable:

```
f(x: INTEGER): [INTEGER -> INTEGER]] =
   LAMBDA (y: INTEGER): x + n * y
```

Although higher-order functions are supported, only the top-level `LAMBDA` may be turned into a defining form. This is not much of an inconvenience, since higher-order functions are not often needed in transition system specifications.

Constant declarations may also be recursive. This is implicit, and the system must be able to determine the measure in order to generate the proper termination obligation:[3]

```
fact(n: NATURAL): NATURAL =
 IF n = 0 THEN 1 ELSE n * fact(n - 1)
```

6.3 Context Declarations

A *ContextDeclaration* provides an abbreviation, e.g., instead of writing

```
lem: LEMMA mycontext{int; 13}!f(3) = mycontext{int; 13}!f(4)
```

[3] As discussed in Section 8.5, this will probably change in the future.

One would write

```
mc: CONTEXT = mycontext{int; 13}
lem: mc!f(3) = mc!f(4)
```

6.4 Assertion Declarations

Assertion expressions allow properties to be stated. In the simplest case these are just boolean-valued expressions, which are thus just logical formulas. The *ModuleModels* form allows properties of modules to be stated. Note that the syntax says nothing about the possible temporal operators; this is defined in a separate context. A *ModuleImplements* assertion M_C `IMPLEMENTS` M_A, says that any possible behavior of M_C is also a behavior of M_A. This allow refinement and abstraction relations to be specified.

AssertionExpression	:=	*ModuleAssertion* | *PropositionalAssertion* | *QuantifiedAssertion* | *Expression*
ModuleAssertion	:=	*ModuleModels* | *ModuleImplements*
ModuleModels	:=	*Module* `|-` *Expression*
ModuleImplements	:=	*Module* `IMPLEMENTS` *Module*
PropositionalAssertion	:=	*PropOp* `(` *AssertionExpression* `,` *AssertionExpression* `)`
		| `NOT (` *AssertionExpression* `)`
QuantifiedAssertion	:=	*Quantifier* `(` *VarDecls* `) :` *AssertionExpression*
PropOp	:=	`AND` | `OR` | `=>` | `<=>`

Chapter 7

Another SAL Example: Mutual Exclusion

We show another example SAL specification: a variant of Peterson's mutual exclusion algorithm [6]. Here the state of the `process` module consists of the controlled variables corresponding to its own program counter `pc1` and boolean variable `x1`, and the observed variables are the corresponding `pc2` and `x2` of the other process. Initially `process` is `sleeping`. The `process` module is parameterized with a boolean `tval` argument.

The `system` is then the asynchronous composition of two processes, where the variables of the `process[TRUE]` have been renamed in order to make them compatible with `process[FALSE]`, i.e., the outputs of one are wired to the inputs of the other.

The main property of this algorithm is assertion `mutex`, which asserts the safety property that in `system`, it is always true that the two processes are not both in their `critical` sections. The assertion language used here is LTL. `G` represents the henceforth modality and `F` represents eventually. Other properties are given, for example `livenessbug1` states the liveness property that it is always possible for `process[FALSE]` to reach its `critical` section. This property is false, because there is no fairness built-in to SAL, so `process[TRUE]` can simply run forever. The same is true for `livenessbug2`. The other liveness properties bring in fairness constraints explicitly, and are provable.

```
peterson: CONTEXT =
BEGIN
  PC: TYPE = {sleeping, trying, critical};

  process [tval : BOOLEAN]: MODULE =
    BEGIN
      INPUT pc2 : PC
      INPUT x2 : BOOLEAN
      OUTPUT pc1 : PC
      OUTPUT x1 : BOOLEAN
      INITIALIZATION pc1 = sleeping
      TRANSITION
        [
```

```
        wakening:
          pc1 = sleeping --> pc1' = trying; x1' = x2 = tval
        []
        entering_critical:
          pc1 = trying AND (pc2 = sleeping OR x1 = (x2 /= tval))
          --> pc1' = critical
        []
        leaving_critical:
          pc1 = critical --> pc1' = sleeping; x1' = x2 = tval
      ]
  END;

  system: MODULE =
     process[FALSE]
     []
     RENAME pc2 TO pc1, pc1 TO pc2,
            x2 TO x1, x1 TO x2
      IN process[TRUE];

  mutex: THEOREM system |- G(NOT(pc1 = critical AND pc2 = critical));

  invalid: THEOREM system |- G(NOT(pc1 = trying AND pc2 = critical));

  livenessbug1: THEOREM system |- G(F(pc1 = critical));

  livenessbug2: THEOREM system |- G(F(pc2 = critical));

  liveness1: THEOREM system |- G(pc2 = trying => F(pc2 = critical));

  liveness2: THEOREM system |- G(pc1 = trying => F(pc1 = critical));

  liveness3: THEOREM system |- G(F(pc1 = trying)) => G(F(pc1 = critical));

  liveness4: THEOREM system |- G(F(pc2 = trying)) => G(F(pc2 = critical));

END
```

Note: the assertions in the THEOREMS are not technically type correct, because the LTL operators `G` and `F` are not defined locally. They are built-in to the SALENV tools descibed in http://sal.csl.sri.com/salenv.html. To make this valid would require defining a LTL context, then including the context name (along with the parameters) in the references to `G` and `F`. In addition, `G` and `F` technically operate on path formulas, so giving them a type that allows them to operate on boolean formulas is a problem. Sections 8.2.1 and 8.3 address these issues.

Chapter 8

Future Work

This language manual and SAL itself are a work in progress.

8.1 SAL as an Intermediate Language

SAL was originally intended to be an intermediate language, but as work progressed it became clear that many users were going to use the language directly, not as an internal representation for some front end. In addition, the desire to create a SAL tool bus, and to keep it language independent, led to the decision to create an abstract syntax in XML, and treat that as the intermediate form. XML was chosen because it is widely used, extensible, and most popular programming languages have direct support for reading and representing XML datas structures.

We have thus defined an abstract syntax in XML by a document type description (DTD), available at `http://sal.csl.sri.com/documentation.html`. The SAL parser (`http://sal.csl.sri.com/salparser.html`) simply reads the concrete syntax and generates an XML file that satisfies the SAL DTD. The separation of the abstract and concrete syntax has many benefits, in that the concrete language may be extended in various ways for convenience, yet map to a more restricted set of data structures, which means that tools do not need to be modified everytime something is added to the concrete language. In addition, users may create their own concrete languages, as long as there is a mapping to the SAL XML abstract syntax.

A general rule followed by the SAL parser is that any transformations done by the parser in creating the abstract structures must, in principle, be invertible. In other words, it should be possible to prettyprint the abstract syntax and get back the original form, ignoring whitespace.

8.2 A SAL Prelude

The language described here has many built-ins, such as `INTEGER`, `AND`, `+`, etc. In principle, these could be defined in a separate context, and imported. This would make the language cumbersome, so instead they were built-in. In our opinion a better choice would be to define these in a prelude, that is automatically imported and provides types, constants, and lemmas. For example, various logics such as CTL, CTL*, and LTL can be defined in the prelude, and even given semantics.

The main advantage of a prelude is that it separates the core language from entities built on the core language. This means that changes to the language can be kept to a minimum, while still allowing new types and constants to be treated as if they were built-in. This is simlar to the separation of the core language of C from its numerous libraries.

Any given SAL tool should be able to read the prelude, and build a symbol table, so it should not be difficult to support.

8.2.1 Libraries, Importings, and Logics

The language defined here may only refer to names outside the context using the fully qualified name. This is helped somewhat with *ContextDeclaration*s, but if a large hierarchy is built up, even this will lead to specifications that are difficult to write and to read. In moving away from the view that this is solely an intermediate language, we feel that the addition of libraries, importings, and logics would be useful, at least in the concrete language.

A library is really just an extension of the idea of a prelude, and allows sets of contexts to be defined in a separate directory, and packaged for broad use and distribution, as with PVS libraries.

Importing a context instance allows the names from that context to be used without a qualifier. There would be restrictions: name conflicts will not be allowed, even if the entities are not comparable. If a referenced name has an associated declaration both in the current context and an imported one, the local one always is used. If a referenced name is common to two separate contexts (including different instances of the same context), then it is an error, and the name must be fully qualified.

Importing a logic is similar, but the idea here is that a logic may be parameterized with the transition system defined by a module, and many instances may be needed for multiple module expressions. A logic declaration would be similar to an importing, but the information needed to instantiate it is derived from the module assertions, for example, a `CTL` context could be defined,

```
ctl{state: TYPE;
    init: [state -> BOOLEAN],
    trans: [[state, state] -> BOOLEAN]} : CONTEXT =
  ...
```

this can then be used as follows:

```
LOGIC ctl
asafety: LEMMA async_bak |- AG(NOT (pc1 = l3 AND pc2 = l3));
```

rather than the error prone and unreadable expanded form:

```
asafety: LEMMA
  async_bak |- ctl{async_bak.STATE;async_bak.INIT,async_bak.TRANS}!
                  AG(NOT (pc1 = l3 AND pc2 = l3));
```

This would address the problem with the peterson specification described in Chapter 7.

Note that in principle a parser for the concrete language can parse the imported contexts, produce name conflict errors, and generate XML files that do not have any importings. This kind of tranformation means that the abstract syntax can be kept minimal, while allowing the concrete syntax to be much more convenient and readable.

8.3 Conversions

The preceding section described a CTL formula. In CTL, `AG` is a predicate transformer, of type `[[STATE -> BOOLEAN] -> [STATE -> BOOLEAN]]`. But of course `NOT` and `AND` are `BOOLEAN` operators, so there is a mismatch. PVS provides a mechanism, called *lambda conversion*, that is very effective in lifting such operators, in this case the result would be as follows:

```
AG(LAMBDA (s: STATE): NOT (pc1(s) = l3 AND pc2(s) = l3))
```

Of course, if SAL was only intended for CTL, this could simply be built-in, but SAL is intended to be logic-independent. For LTL, the formulas are path formulas, not state formulas. In fact, LTL often treats state formulas as path formulas. So a more comprehensive treatment is needed, and conversions look like a reasonable approach.

8.4 Empty Types

As discussed in the `adder` example in Chapter 2, the restriction to nonempty types can actually get in the way of succinct specifications. In the adder case, there is no real problem with having the empty type, it simply means that the `onebitadder` is composed with a module that always skips. Thus in a *MultiSynchronous* composition if the index type is empty, the result is a module with no state variables that always skips. If it is a *MultiAsynchronous* composition, the result is an empty module with no transitions (i.e., it is deadlocked). PVS allows empty types, and there is no logical difficulty. One must, of course, be careful with applying logical rules, in particular those involving quantifiers. For example, one can usually ignore quantifiers whose bound variables do not occur in the underlying expression, but if empty types are allowed, this is unsound. Thus `FORALL (x: T): FALSE` could naively be reduced to `FALSE`, but if the type `T` is empty it is actually vacuously `TRUE`. Also allowing a type to be nonempty means that the declaration of a constant may entail a nonemptiness obligation on the type.

8.5 Recursive Function Termination

In PVS, recursive functions must include a measure, and optionally a well-founded ordering. In earlier discussions of SAL it was thought that functions would be simple enough that the typechecker would always be able to figure out the measure, but this is clearly not true; even the usual definition for GCD requires a measure on the difference of the arguments, and it is not at all clear how a typechecker would be able to determine this. In the future we plan to allow a measure and ordering to be optionall provided by the user.

8.6 State-Dependent Types

Types in SAL are static, but there are situations where having a type that depends on the state is more expressive. In effect, it means that the type can change as the system progresses. The typechecker would generate proof obligations that in every reachable state all state variables satisfy their types. State-dependent types might be useful, for example, in modeling adjustable arrays, where an array may change size dynamically, but it is preferable to prove that a runtime arrays-bound check is not necessary.

Bibliography

[1] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer an d S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *98*, volume 1427, pages 521–525. Vancouver, Canada, June 1998. 2

[2] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991. 2

[3] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993. 2

[4] Ralph Melton and David L. Dill. *Murϕ Annotated Reference Manual*. Computer Science Department, Stanford University, Stanford, CA, March 1993. 2

[5] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*, September 1999. 6

[6] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981. 25

Index

A Technique for Invariant Generation*

To be presented at Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001, Genova, Italy, April 2001.

A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar

SRI International,
333 Ravenswood Ave,
Menlo Park, CA, U.S.A
{tiwari,ruess,saidi,shankar}@csl.sri.com

Abstract. Most of the properties established during verification are either invariants or depend crucially on invariants. The effectiveness of automated formal verification is therefore sensitive to the ease with which invariants, even trivial ones, can be automatically deduced. While the strongest invariant can be defined as the least fixed point of the strongest post-condition of a transition system starting with the set of initial states, this symbolic computation rarely converges. We present a method for invariant generation and strengthening that relies on the simultaneous construction of least and greatest fixed points, restricted widening and narrowing, and quantifier elimination. The effectiveness of the method is demonstrated on a number of examples.

1 Introduction

The majority of properties established during the verification of programs are either invariants or depend crucially on invariants. Indeed, safety properties can be reduced to invariant properties, and to prove progress one usually needs to establish auxiliary invariance properties too. Consequently, the discovery and strengthening of invariants is a central technique in the analysis and verification of both sequential programs and reactive systems, especially for infinite state systems.

Consider, for example, a program with state variables pc and x. The program counter pc is interpreted over the control locations inc and dec, and x is interpreted over the integers. Initially, the program counter pc is set to inc and x to 0. The dynamics of the system is described in terms of the guarded commands:

$$pc = inc \longmapsto x := x + 2;\ pc := dec$$
$$pc = dec \wedge x > 0 \longmapsto x := x - 2;\ pc \in \{inc, dec\}$$

Suppose we are interested in establishing the invariant $pc = inc \rightarrow x = 0$. A naïve proof attempt fails, and consequently, the invariant needs to be strengthened to an inductive invariant $(pc = inc \rightarrow x = 0) \wedge (pc = dec \rightarrow x = 2)$. Such

* The research described in this paper was supported in part by NSF contract CCR-9712383 and DARPA/AFRL contract F33615-00-C-3043.

strengthenings are typically needed in induction proofs. In general, the main principle for proving that a predicate ϕ is an invariant of some program or system S, consists in finding an *auxiliary* predicate ψ such that ψ is stronger than ϕ and ψ is inductive; i.e., every initial state of S satisfies ψ, and ψ is preserved under all transitions. This rule is sound and (relatively) complete. On the other hand, finding a strengthening ψ is not always obvious, and usually requires a microscopic examination of failed verification attempts.

Most approaches for generating and strengthening invariants are based on symbolic computation of the system at hand [4, 10, 15]. The *bottom-up* method performs an abstract forward propagation to compute the set of all reachable configurations, while the *top-down* method starts from an invariant candidate ϕ and performs an abstract backward propagation to compute a strengthened invariant ψ. There is, however, no guarantee for success in exact forward or backward propagation. This may be due either to infinite or unmanageably large configuration spaces or to the failure to detect convergence of the propagation methods altogether. Consequently, approximation techniques such as widening or narrowing [8] are needed to enforce termination of symbolic computation. The basic idea is to accelerate the convergence of symbolic computations in infinite abstract domains.

The framework of abstract interpretation with widening and narrowing as outlined in [8], however, is not immediately applicable to the discovery and strengthening of inductive invariants, since not every over-approximation of an inductive invariant is necessarily an inductive invariant. Our main contributions are: first, we provide an abstract description of the process of *inductive* invariant generation and strengthening based on computing under- and over-approximations of the reachable state set; second, this framework is instantiated with a novel technique based on combining concrete widening and narrowing operators. Our techniques can uniformly be used on a wide class of examples including transition systems where both forward and backward propagation do not converge. We demonstrate the effectiveness of our approach through a variety of examples.

Our algorithm is based on the symbolic computation of a sequence of under- and over-approximations of the reachable state set. These computations rely heavily on the elimination of quantifiers in the underlying theory. Quantifier elimination, however, is not required to return equivalent formulas, since our algorithm tolerates weakened quantifier-eliminated formulas. Whenever the computation of the sequence of under-approximations terminates, we get an inductive invariant. Moreover, since every element in the sequence of decreasing over-approximations is an inductive invariant, our algorithm can be stopped at *any time* and it outputs the best (strongest) inductive invariant computed up to this point. In the example above, our procedure yields the invariant $(pc = inc \rightarrow x = 0) \wedge (pc = dec \rightarrow x = 2)$.[1]

[1] This example can also be handled by some other invariant generation techniques based on forward reachability or abstraction [3, 17].

The approach faces two problems. First, the computation of the sequence of under-approximations usually does not terminate. Second, the computation of the sequence of over-approximations terminates with very weak invariants, in practice. For instance, forward reachability does not converge in case the initial value for x is unspecified in the example above. In order to overcome these problems we add specialized widening and narrowing operators to our algorithm. One of the distinguishing features of our algorithm is the use of unreachable configurations for detecting unreachable strongly connected components and computing corresponding narrowing operators. In this way, our algorithm terminates with the invariant $x > -2$ in case the initial value for x is unspecified in our running example.

The paper is structured as follows. In Section 2 we introduce notation and definitions, Section 3 presents the theoretical framework that is used in Section 4 to obtain a procedure for generating invariants using affirmation and propagation rules along with widening and narrowing. Finally, we conclude in Section 5 with a short investigation of the relationship between invariant generation and abstract interpretation, and comparisons with related work.

2 Preliminaries

Let Σ be a first-order language containing interpreted symbols for standard concrete domains like booleans, integers and reals. Let $\Re$ denote the (first-order) theory of interest over the language Σ. We fix the set $\mathcal{V} = \{x_1, \ldots, x_n\}$ of (typed) variables and denote by $\mathcal{F}$ the set of first-order formulas over Σ with free variables contained in the set $\mathcal{V}$. A *transition system* S is a tuple $(\mathcal{V}, \Theta, \Phi)$, where $\Theta \in \mathcal{F}$ and Φ is a first-order formula over Σ with free variables contained in the set $\mathcal{V} \cup \mathcal{V}'$, where $\mathcal{V}' = \{x'_1, \ldots, x'_n\}$. The formula Θ is called the *initial predicate* and the formula Φ a *transition predicate* of the system S. We shall denote the sequence $x_1, \ldots, x_n$ by $\boldsymbol{x}$ and the sequence $x'_1, \ldots, x'_n$ by $\boldsymbol{x'}$.

A *state* σ of a transition system $\mathrm{S} = (\mathcal{V}, \Theta, \Phi)$ is a mapping from $\mathcal{V}$ to values from the corresponding domains. If ρ is a state, we denote by ρ' the mapping obtained by renaming variables x_i to x'_i in ρ. A formula $\phi(\boldsymbol{x})$ is interpreted as the set $[\![\phi(\boldsymbol{x})]\!]$ of all states σ such that $\Re, \sigma \models \phi(\boldsymbol{x})$. We define the set $\mathit{Reach}(\Phi)(\Theta)$ of states *reachable* from the states represented by Θ via the transition predicate Φ as the smallest set such that (i) $[\![\Theta]\!] \subset \mathit{Reach}(\Phi)(\Theta)$ and (ii) the state $\sigma \in \mathit{Reach}(\Phi)(\Theta)$ whenever $\Re, \rho, \sigma' \models \Phi(\boldsymbol{x}, \boldsymbol{x'})$ for some $\rho \in \mathit{Reach}(\Phi)(\Theta)$. Since the theory $\Re$ is fixed, we shall not mention it explicitly when we talk about satisfiability and validity in $\Re$. Thus, validity in $\Re$ is denoted by $\models$.

A *formula transformer* Γ is a function mapping formulas to formulas. The *strongest postcondition* transformer, denoted by $\mathtt{SP}(\Phi)$, is defined as $\mathtt{SP}(\Phi)(\phi(\boldsymbol{x})) = \exists \boldsymbol{y}.(\Phi(\boldsymbol{y}, \boldsymbol{x}) \wedge \phi(\boldsymbol{y}))$. The formula $\mathtt{SP}(\Phi)(\phi(\boldsymbol{x}))$ denotes the set of states reachable in one step from the set of states represented by ϕ. Similarly, the *weakest precondition* transformer, $\mathtt{WP}(\Phi)$, is defined as $\mathtt{WP}(\Phi)(\phi(\boldsymbol{x})) = \forall \boldsymbol{y}.(\Phi(\boldsymbol{x}, \boldsymbol{y}) \rightarrow \phi(\boldsymbol{y}))$.

A *fixed point* of a formula transformer Γ is a formula ϕ such that $\models \Gamma(\phi) \leftrightarrow \phi$. A formula transformer Γ is *monotonic* if $\models \Gamma(\phi) \rightarrow \Gamma(\psi)$ whenever $\models \phi \rightarrow \psi$. A

least fixed point of Γ, denoted by $\mu\psi.\Gamma(\psi)$, is a fixed point ϕ such that for any other fixed point ψ of Γ, it is the case that $\models (\phi \rightarrow \psi)$. A *greatest* fixed point of Γ, denoted by $\nu\psi.\Gamma(\psi)$, is a fixed point ϕ such that for any other fixed point ψ of Γ, it is the case that $\models (\psi \rightarrow \phi)$. Whenever the transition system $\langle \mathcal{V}, \Theta, \Phi \rangle$ is clear from the context, we define the transformer $\mathcal{I}$ by $\mathcal{I}(\phi) = \mathtt{SP}(\Phi)(\phi) \vee \Theta$. Note that the transformer $\mathcal{I}$ is monotonic. The least fixed point of this operator, $\mu\psi.\mathcal{I}(\psi)$, whenever it exists in the first-order language, represents the set $Reach(\Phi)(\Theta)$ of reachable states.

2.1 Invariants

A formula ϕ is an *S-invariant* if $Reach(\Phi)(\Theta) \subset [\![\phi]\!]$. Thus, an invariant describes an over-approximation of the set of reachable states. An *S-inductive invariant* is a formula ϕ such that (i) ϕ is an S-invariant, and (ii) ϕ is inductive, i.e., $\models \mathtt{SP}(\Phi)(\phi) \rightarrow \phi$. Condition (ii) can be equivalently stated as $\models \phi \rightarrow \mathtt{WP}(\Phi)(\phi)$. In other words, ϕ is an S-inductive invariant if $\models \mathcal{I}(\phi) \rightarrow \phi$. Note that the definition does not require an equivalence, but only an implication.

It is easy to establish that the set of reachable states $Reach(\Phi)(\Theta)$ of a system S represents the strongest (inductive) invariant. By this we mean that if ψ is any other (inductive) invariant, then, $Reach(\mathrm{S})(\Theta) \subset [\![\psi]\!]$. However, note that if ϕ is an inductive invariant, and $\models (\phi \rightarrow \psi)$, then ψ need not be an inductive invariant because ψ might violate condition (ii). For purposes of this paper, we will only be interested in inductive invariants. Thus, we are *not* interested in just obtaining *any* over-approximation of the set of reachable states, but only those that also satisfy condition (ii). This is because the inductive property provides a sufficient local characterization of invariance property, which makes the task of proving easier.

Given a transition system $\mathrm{S} = (\mathcal{V}, \Theta, \Phi)$, the *converse* transition system $\mathrm{S}^{-1} = (\mathcal{V}, \Theta, \Phi^{-1})$ is defined by $\Phi^{-1}(\boldsymbol{x}, \boldsymbol{y}) = \Phi(\boldsymbol{y}, \boldsymbol{x})$. The following well-known theorem says that if none of the initial states is backward reachable from the states represented by ϕ, then $\neg\phi$ is an invariant.

Theorem 1. *Let $S = \langle \mathcal{V}, \Theta, \Phi \rangle$ be a transition system and ϕ an arbitrary formula. If ψ is such that $\models (\mathtt{SP}(\Phi^{-1})(\psi) \vee \phi) \rightarrow \psi$ and the formula $\Theta \wedge \psi$ is unsatisfiable, then $\neg\psi$ is an S-inductive invariant.*

Corollary 1. *If $Reach(\Phi^{-1})(\phi) \cap [\![\Theta]\!] = \emptyset$, then the formula corresponding to the complement of the set $Reach(\Phi^{-1})(\phi)$ is an S-inductive invariant.*

We remark here that although application of the $\mathtt{SP}(\Phi)$ transformer is called "forward propagation", the term "backward propagation" is typically used for the transformer $\mathtt{WP}(\Phi)$. But there is no anomaly here as the transformers $\mathtt{SP}(\Phi^{-1})$ and $\mathtt{WP}(\Phi)$ are duals in the sense that $\mathtt{SP}(\Phi^{-1})(\phi)$ is logically equivalent to $\neg\mathtt{WP}(\Phi)(\neg\phi)$. Hence, Theorem 1 can be stated in terms of $\mathtt{WP}(\Phi)$. It also follows that if formula ϕ is an invariant, then the formula $\nu\psi.\phi \wedge \mathtt{WP}(\Phi)(\psi)$ is an induc-

tive invariant that is a strengthening of ϕ[2]. Similarly, it is easy to see that there is a corresponding connection between the $\mathtt{SP}(\Phi)$ and $\mathtt{WP}(\Phi^{-1})$ transformers.

3 Inductive Invariant Generation

In this section, we discuss the problem of automatically generating some useful inductive invariants for a given transition system. It is a simple observation that the greatest fixed point $\nu\phi.\mathcal{I}(\phi)$, whenever it exists, is an S-inductive invariant.

Lemma 1. *Let $S = \langle \mathcal{V}, \Theta, \Phi \rangle$ be a transition system. Recursively define the sequence of formulas $\phi_0, \phi_1, \ldots$, as follows.*

$$\phi_0 = \mathtt{true} \qquad \phi_{i+1} = \mathtt{SP}(\Phi)(\phi_i) \vee \Theta$$

Then, every formula ϕ_i is an S-inductive invariant. Furthermore, every formula ϕ_i in the above sequence can be decomposed as $\psi_i \vee \chi_i$, where

$$\begin{array}{ll} \psi_0 = \mathtt{false} & \psi_{i+1} = \mathtt{SP}(\Phi)(\psi_i) \vee \Theta \\ \chi_0 = \mathtt{true} & \chi_{i+1} = \mathtt{SP}(\Phi)(\chi_i). \end{array}$$

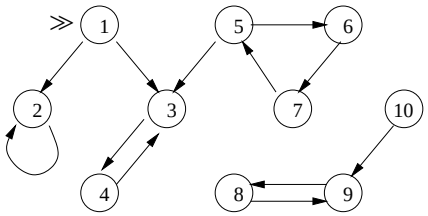
The sequence $\psi_0, \psi_1, \ldots$, represents iterations in a least fixed point computation of the $\mathcal{I}$ transformer. The sequence $\chi_0, \chi_1, \ldots$, represents the greatest fixed point component. The formulas ψ_i provide successive under-approximations of the set $Reach(\Phi)(\Theta)$ of reachable states. The formulas ϕ_i are inductive over-approximations. The sequence $\psi_0, \psi_1, \ldots$, usually does not terminate, whereas the sequence $\phi_0, \phi_1, \ldots$, often terminates with very weak invariants.

It should be observed here that the greatest fixed point of the $\mathtt{SP}(\Phi)(\_) \vee \Theta$ transformer characterizes states σ such that there exists a backward path starting from σ which is either infinite, or contains some initial state. In case of finite state transition systems, this is exactly the set of states that either belong to a strongly connected component, or, that are reachable from either some initial state or some strongly connected component. Hence, the greatest fixed point may not be the strongest S-inductive invariant even in the case of finite systems. Despite its shortcomings, this simple method is attractive since (i) we do not need to detect that the iterations have converged[3], and (ii) every formula ϕ_i is an S-inductive invariant. Detecting convergence is difficult as it involves deciding if $\models \phi_i \leftrightarrow \phi_{i+1}$.

Example 1. Consider the transition system over ten states presented in Figure 1.

[2] It follows from this duality that the the least (greatest) fixed point iterations of $\mathtt{SP}(\Phi^{-1}) \vee \phi$ are logically equivalent to the negations of the greatest (least) fixed point iterations of $\mathtt{WP}(\Phi) \wedge \neg\phi$.

[3] If ϕ is an S-invariant, then every iteration in the greatest fixed point computation of $\mathtt{WP}(\Phi)(\_) \wedge \phi$ is also an S-invariant. But, if ϕ is inductive, then this method yields ϕ.



States are represented by nodes with integer labels and transitions are represented by edges. State 1 is the initial state.
Clearly, the set of reachable states is the set

$$\{1, 2, 3, 4\}.$$

The greatest fixed point of the $\mathtt{SP}(\Phi) \vee \Theta$ is the set consisting of states

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

Fig. 1. A finite state transition system.

3.1 Widening and Narrowing

In the case when the state space is either infinite, or finite but too large, the symbolic computation of (greatest or least) fixed points of various transformers is restricted by the finite space and time resources available. A well-known solution to this problem is the use of *widening* and *narrowing* to respectively enhance the least and greatest fixed point computation (with gains obtained both in terms of space and time).

A *widening* operator $\bigtriangledown : \mathcal{F} \times \mathcal{F} \mapsto \mathcal{F}$ is a function such that for all formulas $\phi, \phi' \in \mathcal{F}$, $\models (\phi \vee \phi') \rightarrow \bigtriangledown(\phi, \phi')$. Similarly, a *narrowing* operator $\bigtriangleup : \mathcal{F} \times \mathcal{F} \mapsto \mathcal{F}$ is a function such that for all formulas $\phi, \phi' \in \mathcal{F}$, $\models \bigtriangleup(\phi, \phi') \rightarrow (\phi \wedge \phi')$. Thus, logical disjunction $\vee$ is a trivial widening operator, and logical conjunction $\wedge$ is a trivial narrowing operator.

The definitions of widening and narrowing are slightly different from the standard ones [8, 9]. First, we do not include any conditions to guarantee that increasing (decreasing) sequences are transformed to finite, hence converging, increasing (decreasing) sequences by widening (narrowing). Secondly, in the case of narrowing, the standard definition requires that whenever $\phi' \rightarrow \phi$, the formula $\bigtriangleup(\phi, \phi')$ is such that $\phi' \rightarrow \bigtriangleup(\phi, \phi')$ and $\bigtriangleup(\phi, \phi') \rightarrow \phi$. In our definition, $\bigtriangleup(\phi, \phi')$ is stronger than both ϕ and ϕ' as our interest is in the use of narrowing to obtain *under*-approximations of the greatest fixed point. But we have to be careful so as to not eliminate any reachable states by overly aggressive under-approximation, see Lemma 3.

A particularly simple narrowing operator, denoted by $\bigtriangleup(\psi)$, is defined by $\bigtriangleup(\psi)(\phi, \phi') = \phi \wedge \phi' \wedge \psi$, where ψ is an arbitrary formula. Similarly, we can define $\bigtriangledown(\psi)(\phi, \phi') = \phi \vee \phi' \vee \psi$. Since we are interested in generating inductive invariants, it turns out that in order to guarantee correctness, we can use any arbitrary widening operator, but not any narrowing operator.

Lemma 2. *[Upward iteration sequence with widening] Let $\psi_0, \psi_1, \ldots,$ be a sequence of formulas such that ψ_0 is Θ, and for every $i > 0$, either*

(i) ψ_i is $\mathtt{SP}(\Phi)(\psi_{i-1}) \vee \psi_{i-1}$, or

(ii) ψ_i is $\bigtriangledown(\alpha_i)(\psi_{i-2}, \psi_{i-1})$, where α_i is any arbitrary formula.

Then, if for some $n > 0$, $\models \mathtt{SP}(\Phi)(\psi_n) \rightarrow \psi_n$, then the formula ψ_n is an S-inductive invariant.

Lemma 3. *[Downward iteration sequence with narrowing] Let $\phi_0, \phi_1, \ldots,$ be a sequence of formulas such that ϕ_0 is* `true`*, and for every $i > 0$, either*

(i) ϕ_i is $\mathtt{SP}(\Phi)(\phi_{i-1}) \vee \Theta$, or

(ii) ϕ_i is $\triangle(\beta_i)(\phi_{i-2}, \phi_{i-1})$, where β_i is some S-inductive invariant.

Then, for every i, ϕ_i is an S-inductive invariant[4] such that $\models \phi_i \rightarrow \beta_i$.

Lemma 3 extends the greatest fixed point iterations in Lemma 1 by a narrowing operator. Similarly, Lemma 2 extends the least fixed point computation that is hidden inside the iterations in Lemma 1 by a widening operator.

We obtain the formula β_i used in Lemma 3 by identifying strongly connected components consisting of unreachable states. This is achieved using backward propagation from an unreachable state, as outlined in Theorem 1. These unreachable states are not automatically eliminated by the greatest fixed point computation outlined in Lemma 1. Furthermore, an S-inductive invariant obtained using Lemma 2 can be used in Step (ii) of Lemma 3. Thus, Lemma 3 gives a method for systematically strengthening known invariants.

Example 2. Following up on Example 1, let $N = \{1, 2, \ldots, 10\}$ denote the set of all states. In order to strengthen the over-approximation, viz. $N - \{10\}$, of the set of reachable states obtained via the greatest fixed point computation, we can try removing certain states. But if we remove a subset of states that is not strongly connected, the subsequent fixed point computation may no longer be monotonic, and could fail to converge.

For instance, removing state 5 from the above set gives a new set $N_1 = N - \{5, 10\}$. Now, $\mathtt{SP}(\Phi)(\phi_{N_1}) \vee \Theta$, where ϕ_{N_1} is the characteristic predicate of N_1, represents the set $N_2 = N - \{6, 10\}$. Clearly, $N_2 \not\subseteq N_1$, and hence the sequence of formulas obtained in the greatest fixed point computation is no longer monotonic. Note that all formulas in the sequence are invariants, but they are not inductive.

In order to identify unreachable states, we note that if we start with the set $N_3 = \{7, 8\}$, and we assign ϕ in Theorem 1 to the characteristic predicate ϕ_{N_3} of N_3. The least fixed point of $\mathtt{SP}(\Phi^{-1}) \vee \phi_{N_3}$ represents the set $N_4 = \{5, 6, 7, 8, 9, 10\}$. Now, since the formula $\Theta \wedge \phi_{N_4}$ is unsatisfiable (i.e. the set $\{1\} \cap N_4 = \emptyset$), it follows from Theorem 1 that the set $N_5 = \{1, 2, 3, 4\}$ represented by $\neg\phi_{N_4}$ is an S-inductive invariant.

4 An Any-time Algorithm for Generating Inductive Invariants

The transition predicate Φ of a transition system $\mathrm{S} = (\mathcal{V} = \{x_1, \ldots, x_n\}, \Theta, \Phi)$ is typically specified using a finite set of *guarded transitions*, where a guarded

[4] Note that the lemma also holds if we drop the word "inductive" from the statement.

transition consists of a guard $\gamma \in \mathcal{F}$, and a finite set of assignments $\{x_1 := e_1(\boldsymbol{x}), \ldots, x_n := e_n(\boldsymbol{x})\}$. A guarded transition τ is written as

$$\gamma \ \longmapsto \ x_1 := e_1(\boldsymbol{x}); \ldots; x_n := e_n(\boldsymbol{x})$$

where e_i is some expression with free variables in the set $\boldsymbol{x}$. We shall also use the compact notation $\boldsymbol{x} := e(\boldsymbol{x})$ to represent the above assignments.

A typical specification of a guarded transition system contains at least one control variable, usually the program counter $pc \in \{x_1, \ldots, x_n\}$, which takes values from a finite set, say $\{1, \ldots, p\}$. Control states are defined by formulas of the form $pc = i$, $i \in \{1, \ldots, p\}$. This transition system then has p different control states. Additionally, we assume that the source states of each guarded transition belong to some *fixed* source control state, $pc = i$, (and similarly for the target states) so that each transition τ can be written as

$$pc = i \wedge \gamma \ \longmapsto \ \boldsymbol{x} := e(\boldsymbol{x}); pc := j$$

where $\boldsymbol{x}$ denotes variables in $\mathcal{V} - \{pc\}$. In this case, we define $src(\tau) = i$ and $tgt(\tau) = j$. By $\Phi_\tau(\boldsymbol{x}, \boldsymbol{x'})$, we denote the formula $\gamma(\boldsymbol{x}) \wedge \boldsymbol{x'} = e(\boldsymbol{x})$. If $\mathcal{T}$ is a set of such transitions, then the transition predicate Φ is itself defined by $\bigvee_{\tau \in \mathcal{T}} pc = src(\tau) \wedge pc' = tgt(\tau) \wedge \Phi_\tau$. Similarly, we assume that $\models \Theta \rightarrow pc = 1$.

Whenever such a decomposition of the state space into finitely many control states is available such that every transition has a unique source and target control state, the S-invariant can be maintained as a conjunction of local invariants indexed by the control locations. We assume that every formula is represented as an array of formulas indexed by integers $\{1, \ldots, p\}$. Given an S-inductive invariant φ (as an array of formulas), and a transition predicate Φ, the function `propagation`$(\Theta, \Phi, \varphi, k)$ returns the strengthened S-inductive invariant $\mathcal{I}^k(\varphi)$.

```
function propagation(Θ, Φ, φ, k) {
    let Θ be pc = 1 ∧ Θ';
    for k iterations do: for every i in parallel do {
        T_i := {τ ∈ T : tgt(τ) = i};
        φ[i] := { ⋁_{τ∈T_i} SP(Φ_τ)(φ[src(τ)]) ∨ Θ'   if i = 1
                { ⋁_{τ∈T_i} SP(Φ_τ)(φ[src(τ)])        if i ≠ 1 };
        φ[i] := ℜ-simplify(φ[i]);
    }
    return(φ);
}
```

The function ℜ-`simplify` performs quantifier-elimination and simplification in the theory $\Re$ and is described in Section 4.2.

Lemma 4. *Let $S = (\mathcal{V}, \Theta, \Phi)$ be a transition system and let φ_0 be an array of formulas initialized to* `true`*. Let φ_k denotes the array* `propagation`$(\Theta, \Phi, \varphi_0, k)$ *of formulas (assuming* ℜ-`simplify` *always returns equivalent formulas), and ϕ_k be as defined in Lemma 1. Then, for all $k \geq 0$, $\models \phi_k \leftrightarrow \bigwedge_{i=1}^{p} (pc = i \rightarrow \varphi_k[i])$. Consequently, the formula $\bigwedge_{i=1}^{p} (pc = i \rightarrow \varphi_k[i])$ is an S-inductive invariant, for every k.*

Notice that the formula $\bigwedge_{i=1}^{p}(pc = i \rightarrow \varphi[i])$ is equivalent to the formula $\bigvee_{i=1}^{p}(pc = i \wedge \varphi[i])$ under the assumption that $\bigvee_{i=1}^{p}(pc = i)$. The computations outlined in other lemmas and theorems can be suitably cast in terms of local invariants at control locations.

4.1 Combining SP(Φ) and SP(Φ^{-1}) iterations

The basic algorithm for the automatic generation of inductive invariants consists of affirmation and propagation steps—the essence of which is captured in Lemma 1 and function `propagation`. In order to get stronger invariants, we propose the use of narrowing and widening.

The function `widening`(ϑ, φ, k) starts with a given under-approximation ϑ of the set of reachable states, and widens it using a subformula α of the over-approximation φ. If this widening yields an S-inductive invariant (see Lemma 2) in k propagation steps, then the function returns this invariant, otherwise it just returns `true`[5].

```
function widening(ϑ, φ, k) {
    χ := ϑ;
    choose j ∈ {1,...,p} and a formula α s.t.
        φ[j] is of the form φ' ∨ α, and ϑ[j] ∧ α is satisfiable;
    χ[j] := χ[j] ∨ α;                          /* widening */
    χ := propagation(Θ, Φ, χ, k);
    if (⊨ propagation(Θ, Φ, χ, 1)[i] → χ[i] for all i)
        return(χ);                             /* new invariant */
    return(true);
}
```

Lemma 5. *For any value of the constant k, if χ denotes the array of formulas returned by* `widening`(ϑ, φ, k)*, then the formula $\bigwedge_{i=1}^{p} pc = i \rightarrow \chi[i]$ is an S-inductive invariant.*

Strongly connected components of unreachable states are detected using backward propagation, and if successful, this information is used for strengthening the current invariant. The subroutine `narrowing`(ϑ, φ, k) chooses a subformula β of the over-approximation φ which could possibly represent unreachable states. Thereafter, it computes the set of states that are backward reachable from the conjectured unreachable states β and if we successfully terminate without intersecting Θ (see Theorem 1), then we again have an S-inductive invariant.

```
function narrowing(ϑ, φ, k) {
   choose j ∈ {1,...,p} and a formula β s.t.
       φ[j] is of the form φ' ∨ β, and ϑ[j] ∧ β is unsatisfiable;
   χ := propagation(pc = j ∧ β, Φ^-1, false, k);
```

[5] We shall overload `true` (`false`) to also denote arrays in which every element is `true` (`false`), and use assignments between arrays to mean element-wise copying.

```
    if (|=propagation(pc = j ∧ β, Φ^{-1}, χ, 1)[i] → χ[i] for all i)
       if (|= ¬(χ ∧ Θ))
          return(Invariant(¬χ));
       else if (χ ∧ Θ is satisfiable)        /* β is reachable */
          return(Reachable(pc = j ∧ β));
    return(Invariant(true));
 }
```

The return value `Reachable(`ψ`)` of the function `narrowing(`ϑ, φ, k`)` says that the states represented by ψ are reachable, and the return value `Invariant(`ψ`)` denotes that the formula represented by ψ is an inductive invariant.

Lemma 6. *For any value of the constant k, if the function* `narrowing(`ϑ, φ, k`)` *returns* `Reachable(`ψ`)`*, then* $[\![\psi]\!] \subset \mathit{Reach}(\Phi)(\Theta)$*. Similarly, for any value of the constant k, if* `narrowing(`ϑ, φ, k`)` *returns* `Invariant(`ψ`)`*, then the formula* $\bigwedge_{i=1}^{p}(pc = i \rightarrow \psi[i])$ *is an S-inductive invariant.*

Finally, we outline a procedure that uses the various functions described above by combining the least fixed point and greatest fixed point computations with narrowing and widening. In the procedure, the formula ψ always stores an under-approximation of the set of reachable states, and the formula ϕ always stores an S-inductive invariant. The procedure essentially consists of doing one of four different steps—(i) Augmenting ψ using `propagation(`Θ, Φ, ψ, k`)`, where k is some constant; (ii) Strengthening the current invariant ϕ using the function `propagation(`Θ, Φ, ϕ, k`)`; (iii) Use of widening on the under-approximation for generating an invariant; and, (iv) Use of narrowing to detect and eliminate unreachable states from the over-approximation.

```
/* Given: S = (V, Θ, Φ), a transition system with p control states.
   The transition predicate Φ is indexed by guarded transitions.
   k is an upper bound on the number of iterations. */
Procedure InvGen:
   φ, ψ: Array [1...p] of formula
   Initialization:
       φ := true;
       ψ := false;
   repeatedly do the following {{
       ψ := propagation(Θ, Φ, ψ, k);
       if (|= propagation(Θ, Φ, ψ, 1)[i] → ψ[i] for all i)
           φ := ψ; terminate the program;
   } OR {
       φ := propagation(Θ, Φ, φ, k);
   } OR {
       φ := φ ∧ widening(ψ, φ, k);
   } OR {
       if (narrowing(ψ, φ, k) returns Reachable(β))
           ψ[j] := ψ[j] ∨ χ where β is pc = j ∧ χ;
```

```
        else (assuming narrowing returns Invariant(β))
            φ[i] := φ[i] ∧ β[i] for all i;
    }}
```

Theorem 2. *Let ϕ be the array of formulas in the procedure* `InvGen`*. Then, at any stage of the procedure, the formula $\bigwedge_i(pc = i \rightarrow \phi[i])$ is an S-inductive invariant.*

Our procedure does not consider the control structure of the transition graph to generate invariants. Though specific control structures, like loops, are not relevant for correctness of the basic procedure, they can be important in choosing specific points for widening or narrowing [6]. We wish to point out that the procedure is tolerant to theorem proving failures and only assumes a refutationally complete prover. In particular, note that the satisfiability test in `widening` can be eliminated.

4.2 Quantifier Elimination and Simplification

We remark here that implementation of propagation requires elimination of existential quantifiers. The existential quantifier in $\mathtt{SP}(\Phi^{-1})(\phi)$ and the universal quantifier in $\mathtt{WP}(\Phi)(\phi)$ can both be easily eliminated using substitutions. The quantifiers in $\mathtt{SP}(\Phi)(\phi)$ and $\mathtt{WP}(\Phi^{-1})(\phi)$ cannot be eliminated so easily in general. But in special cases, for instance when the transition is "reversible" (for example, the effect of assignment $x := x + y$ can be reversed by the assignment $x := x - y$), quantifier elimination reduces to substitution again. In cases where exact quantifier elimination is not possible, we can still get a correct procedure using a quantifier elimination procedure that returns a "weaker" formula, i.e., we do *not* need an equivalence preserving quantifier elimination procedure.

Let $\Re$-`simplify` be a function such that $\models \phi \rightarrow \Re\text{-}\mathtt{simplify}(\phi)$. We shall denote the formula $\Re\text{-}\mathtt{simplify}(\phi)$ by $\overline{\phi}$ in the next theorem.

Theorem 3. *Let $\psi_0, \psi_1, \ldots, \psi_i$ be an upward iteration sequence with widening and $\phi_0, \phi_1, \ldots, \phi_i$ be a downward iteration sequence with narrowing (see Lemmas 2 and 3). Then the sequence $\psi_0, \psi_1, \ldots, \psi_i, \overline{\psi_i}$ is also an upward iteration sequence with widening. Similarly, the sequence $\phi_0, \phi_1, \ldots, \phi_{i-1}, \phi_i'$, where ϕ_i' is $\phi_{i-1} \wedge \overline{\phi_i}$, is also a downward iteration sequence with narrowing.*

Note that the formula ϕ_i' in Theorem 3 can be seen as results of "narrowing" in the sense of [9]. Theorem 3 makes it possible for simple (and possibly incomplete) quantifier elimination procedures to suffice for our purposes. For instance, when it is not possible to eliminate the existential quantifier from $\exists x.p(x) \wedge q(x)$, we could weaken this to $\exists x.p(x) \wedge \exists x.q(x)$ and perform quantifier elimination on atomic formulas. With suitable modifications as outlined in Theorem 3, our procedure continues to be correct. In fact, such simplifications help in the convergence of the iterations as well.

Finally, as pointed out in Lemma 1, implementation of the above procedure can be optimized by combining the arrays ψ and ϕ into a single array, say φ.

If individual formulas $\varphi[i]$ are always stored in disjunctive normal form, then we can distinguish the disjuncts that would appear in $\psi[i]$ by marking them. In this way, a single propagation step can be used to update both ψ and ϕ. The implementation of the above procedure is being done in the framework of SAL [1], which is a collection of different tools for analyzing concurrent systems.

4.3 Illustrative Examples

We shall provide certain simple examples to illustrate the procedure. The theory of interest is the theory of linear arithmetic, and we assume that we have an exact quantifier elimination procedure.

Example 3. Consider the example outlined in Section 1. In this case, the least fixed point sequence converges in two steps. In particular, we obtain the invariant $pc = inc \rightarrow x = 0 \wedge pc = dec \rightarrow x = 2$.

Example 4. A simplified version of the Bakery mutual exclusion protocol $\mathrm{S} = (\mathcal{V}, \Theta, \Phi)$ for two processes $p1$ and $p2$ accessing a critical section cs is given by $\mathcal{V} = \{y1 : int, y2 : int, pc1 : \{1,2,3\}, pc2 : \{1,2,3\}\}$, Θ is $pc1 = 1 \wedge pc2 = 1 \wedge y1 = 0 \wedge y2 = 0$, and Φ is defined by the following set of guarded transitions:

$$
\begin{array}{rcll}
pc1 = 1 & \longmapsto & y1 := y2 + 1; pc1 := 2; & // \text{ p1: try} \\
pc1 = 2 \wedge (y2 = 0 \vee y1 \leq y2) & \longmapsto & pc1 := 3; & // \text{ p1: enter cs} \\
pc1 = 3 & \longmapsto & y1 := 0; pc1 := 1; & // \text{ p1: exit cs} \\
pc2 = 1 & \longmapsto & y2 := y1 + 1; pc2 := 2; & // \text{ p2: try} \\
pc2 = 2 \wedge (y1 = 0 \vee y2 < y1) & \longmapsto & pc2 := 3; & // \text{ p2: enter cs} \\
pc2 = 3 & \longmapsto & y2 := 0; pc2 := 1; & // \text{ p2: exit cs}
\end{array}
$$

Since this system has an infinite number of reachable states, the least fixed point computation sequence does not converge. We choose to define 9 control locations based on the values of $pc1$ and $pc2$ variables, and we shall use the notation $\phi[i,j]$ to denote the current invariant at control location $pc1 = i \wedge pc2 = j$. After a few iterations, the greatest fixed point iterations yield a formula ϕ, with the following three local invariants (due to space restrictions, we are not writing down the complete formula here):

$$
\begin{array}{l}
\phi[3,1] : y2 = 0 \\
\phi[3,2] : (y2 = y1 + 1) \vee (y1 = 1 \wedge y2 = 0) \\
\phi[3,3] : (y1 = 0 \wedge y2 = 1) \vee (y1 = 1 \wedge y2 = 0)
\end{array}
$$

The disjunct β, defined as $y1 = 0 \wedge y2 = 1$, in control location $pc1 = 3 \wedge pc2 = 3$ can be conjectured to be unreachable (as the formula $\psi[3,3]$ in the least fixed point iterations is always `false`) and for a suitable choice of k, the formula $\chi := \texttt{propagation}(pc1 = 3 \wedge pc2 = 3 \wedge \beta, \Phi^{-1}, \texttt{false}, k)$ contains the following strongly connected set of unreachable states,

$$
\begin{array}{lll}
\chi[3,3] : y1 = 0 & \chi[3,2] : y1 = 0 & \chi[2,3] : y1 = 0 \\
\chi[3,1] : y1 = 0 & \chi[2,2] : y1 = 0 & \chi[2,1] : y1 = 0
\end{array}
$$

Similarly, we can eliminate the other possibility ($y1 = 1 \wedge y2 = 0$) at control location $pc1 = 3 \wedge pc2 = 3$. This proves mutual exclusion. We can also use a single widening step to obtain an inductive invariant strong enough to prove mutual exclusion. Note that it was pointed out in [5] that the computation of $\nu\phi.(\texttt{WP}(\Phi)(\phi) \wedge (pc1 = 3 \wedge pc2 = 3 \rightarrow \texttt{false}))$ terminates in a finite number of steps and yields an invariant that proves mutual exclusion.

Example 5. Consider the following transitions:

$$\begin{aligned} pc = 1 &\longmapsto x := x + 2; y := y + 2; pc := 2; \\ pc = 2 &\longmapsto x := x - 2; y := y + 2; pc := 1; \end{aligned}$$

with initial state predicate $pc = 1 \wedge x = 0 \wedge y = 0$. Assuming that the variables x and y are declared to be integers, neither the least fixed point sequence, nor the greatest fixed point sequence converges. After a few iterations for computing the greatest fixed point, the formula ϕ we obtain is:

$$\begin{aligned} pc = 1 \rightarrow (x = 0 \wedge y = 0) \vee (x = 0 \wedge y = 4) \vee (x \geq 0 \wedge y \geq 8) \\ pc = 2 \rightarrow (x = 2 \wedge y = 2) \vee (x = 2 \wedge y = 6) \vee (x \geq 2 \wedge y \geq 10) \end{aligned}$$

The predicate $\geq$ can be replaced by the predicates $=$ and $>$. Now, the disjunct β can be chosen as $x > 0 \wedge y \geq 8$ and it can be conjectured to be unreachable. The formula $\texttt{propagation}(pc = 1 \wedge \beta, \Phi^{-1}, \texttt{false}, 2)$ contains the following strongly connected set of unreachable states,

$$pc = 1 \rightarrow x > 0 \wedge y \geq 8 \qquad pc = 2 \rightarrow x > 2 \wedge y \geq 6$$

Conjunction of the negation of this formula with the original invariant ϕ gives the following new invariant,

$$\begin{aligned} pc = 1 \rightarrow (x = 0 \wedge y = 0) \vee (x = 0 \wedge y = 4) \vee (x = 0 \wedge y \geq 8) \\ pc = 2 \rightarrow (x = 2 \wedge y = 2) \vee (x = 2 \wedge y = 6) \vee (x = 2 \wedge y \geq 10) \end{aligned}$$

As before, in this case again widening can also be used to obtain a similar invariant.

5 Related Work and Concluding Remarks

Early work [10, 12] on generating invariant for sequential programs has been extended to the case of reactive systems in [2, 5, 11, 13, 16]. These methods are usually based on the propagation of invariants through the control structure of the different components and by combining local invariants of each component to construct global invariants of the system.

Forward and backward propagation using operators $\texttt{SP}(\Phi)$ and $\texttt{WP}(\Phi)$ is also used in [5] as the basic technique for generating invariants. In addition, over-approximations such as the convex hull of the union of polyhedra, are used for

widening fixed point computations. Our approach differs in that we consider simultaneous forward and backward propagation for computing both lower and upper bounds of the reachable state sets. These bounds are also used for computing suitable narrowing and widening operators. The combination of these techniques usually yields much stronger invariants. Moreover, our algorithm is an any-time algorithm, in the sense that it can be interrupted at any time to yield the most refined inductive invariant computed up to the point of interruption.

The method of generalized reaffirmed invariance and propagation was introduced in [2] and is based on *affirming* local invariants of the form $\mathtt{SP}(\Phi(\tau))(\mathtt{true})$ and *propagating* these local invariants along all transitions. This process of affirmation and propagation, however, is performed only in the special case when all the existential quantifiers arising in the process are trivial, i.e., when the quantified variables do not occur in the rest of the formula; the *twos* example in the introduction does not possess this property. The technique presented in [2] also uses information about the control transition graph, especially knowledge about cycles and how variables are manipulated in the cycle transitions, to generate stronger invariants. In some cases, these stronger local invariants can be generated by repeated propagation (in the stronger sense defined in this paper). In general, however, the detection of unreachable cycles is crucial, as outlined in Theorem 1.

Techniques based on abstraction have also been proposed for generating invariants [3,14]. It appears attractive to first create (finite) abstractions for large programs and then to use standard propagation techniques to obtain the set of states reachable in the abstract system. This set can then be concretized to obtain invariants of the concrete system. Abstraction can be cast as a special widening strategy in our procedure. More specifically, let (α, γ) be an abstraction and concretization pair (Galois connection) for a transition system $\mathrm{S} = (\mathcal{V}, \Theta, \Phi)$. Let $\mathrm{S}_a = (\mathcal{V}_a, \Theta_a, \Phi_a)$ denote the abstract transition system. If

$$\psi_a^{(0)}, \psi_a^{(1)}, \psi_a^{(2)}, \ldots$$

is a least fixed point computation on the abstract transition system S_a, then one obtains a corresponding fixed point computation with widening on the concrete system

$$\psi^{(0)}, \psi^{(1)}, \psi^{(1')}, \psi^{(2)}, \psi^{(2')}, \ldots$$

as follows: the formula $\psi^{(i)}$ is $\mathtt{SP}(\Phi)(\psi^{(i-1')}) \vee \psi^{(i-1')}$ (Step (i) of Lemma 2), and $\psi^{(i')}$ is $\psi^{(i)} \vee \gamma(\alpha(\psi^{(i)}))$ (Step (ii) of Lemma 2). Now, if $\models \gamma(\psi_a^{(i)}) \leftrightarrow \psi^{(i')}$, then it is also the case that $\models \gamma(\psi_a^{(i+1)}) \leftrightarrow \psi^{(i+1')}$. Thus, the fixed point computation on the abstract transition system can be suitably captured in the concrete system. We shall not prove this claim here, but refer to [9] for a similar result.

Note that the set of generated invariants is restricted to the ones expressible in the language of the theory $\Re$. A program that performs multiplication by repeated addition, for example, never uses the multiplication operator, but any expression that describes the set of reachable states typically would use the multiplication operator.

In summary, we present a technique for generation of inductive invariants using a combination of least and greatest fixed point computations of the forward and backward propagation operators. With obvious modifications, the results can be used to strengthen invariants. Thus, any technique for generation of invariants, inductive or not, can be incorporated with the techniques in this paper.

Acknowledgements. We would like to thank S. Bensalem, S. Owre, Y. Lakhnech, J. Rushby, J. Sifakis, and the referees for their helpful comments.

References

[1] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, 2000. Available at http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/.

[2] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15:75–92, 1999.

[3] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proc. of the 9th Conference on Computer-Aided Verification, CAV'98*, LNCS. Springer Verlag, June 1998.

[4] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in LNCS, pages 323–335. Springer-Verlag, 1996.

[5] N. Bjørner, A. Browne, and Z. Manna. Automatic Generation of Invariants and Intermediate Assertions. *Theoretical Computer Science*, 1997.

[6] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the Intl Conf on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 128–141. Springer Verlag, 1993.

[7] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, January 1977.

[9] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proc. of the 4th Intl. Symposium on Programming Language Implementation and Logic Programming* (*PLILP '92*), volume 631 of *LNCS*, pages 269–295, Berlin, 1992. Springer-Verlag.

[10] S. M. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, March 1975.

[11] S. Graf and H. Saïdi. Verifying invariants using theorem proving. In *Conference on Computer Aided Verification CAV'96*, LNCS 1102, Springer Verlag, 1996.

[12] S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, April 1976.

[13] L. Lamport. The 'Hoare logic' of concurrent programs. In *Acta Informatica 14*, pages 21–37, 1980.

[14] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1), January 1995.

[15] Z. Manna and A. Pnueli. *The Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.

[16] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[17] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *Computer-Aided Verification, CAV '99*, Trento, Italy, July 1999.